# Implementation of a MPEG 1 Layer I Audio Decoder with Variable Bit Lengths

A thesis submitted in
fulfilment of the requirements of the
degree of Master of Engineering

23 September 2008

Damian O'Callaghan

B. Eng. (Electrical/Electronic)

School of Electrical & Computer Systems Engineering

Science, Engineering and Technology Portfolio

RMIT University

# Abstract

One of the most popular forms of audio compression is MPEG (Moving Picture Experts Group). By using a VHDL (Very high-speed integrated circuit Hardware Description Language) implementation of a MPEG audio decoder and varying the word length of the constants and the multiplications used in the decoding process, and comparing the error, the minimum word length required can be determined. In general, the smaller the word length, the smaller the hardware resources required.

This thesis is an investigation to find the minimum bit lengths required for each of the four multiplication sections used in a MPEG Audio decoder, that will still meet the quality levels specified in the MPEG standard. The use of the minimum bit lengths allows the minimum area resources of a FPGA (Field Programmable Gate Array) to be used.

A FPGA model was designed that allowed the number of bits used to represent four constants and the results of the multiplications using these constants to vary. In order to limit the amount of data generated, testing was restricted to a single channel of audio data sampled at a frequency of 32kHz. This was then compared to the supplied C model distributed with the MPEG Audio Standard.

It was found that for the MPEG audio coder to be fully compliant with the standard the bit lengths of the constants and the multiplications could be reduced by 75% and to be partial compliant with the standard, the bit lengths of the constants and the multiplications could be reduced by up to 82%.

An implementation of a MPEG audio decoder in VHDL has the advantage of specific hardware, optimised, for all the different complex mathematical operations thereby reducing the repetitive operations and therefore power consumption and the time required performing these complex operations.

# Contents

# List of Figures

# List of Tables

# Declaration

I certify that except where due acknowledgment has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Damian O'Callaghan
2 October 2008

# Glossary

**AAC** – Advanced Audio Coding

**AC-3** - Adaptive Transform Coder 3

**ADPCM** – Adaptive Differential Pulse Coded Modulation

**CD** - compact disc

**CELP** – Code Excited Linear Prediction

**CRC** - Cyclic Redundancy Check

**DAB** - Digital Audio Broadcasting

**DCT** - Discrete Cosine Transform

**DPCM** – Differential Pulse Coded Modulation

**DSP** – Digital Signal Processor

**DTS** - Digital Theatre Sound

**DVD** – Digital Versatile Disc

**FFT** – Fast Fourier Transform

**FPGA** - Field Programmable Gate Arrays

**IDCT** – Inverse Discrete Cosine Transform

**IEC** – International Electrotechnical Commission of standards

**ISO** - International Standards Organisation

**kbit/second** – kilobits per second

**kHz** - kilohertz

**MPEG** - Moving Picture Experts Group

**NOS** – number of slots

**PC** - personal computer

**PCM** – Pulse Coded Modulation

**VCD** - Video Compact Disc

**VHDL** - Very high speed integrated circuit Hardware Description Language

**VHS** – Video Home System

# Chapter 1. Introduction, Scope and Outcomes

## 1.1 Overview

Audio compression and decompression algorithms are useful because they allow large amounts of audio data to be shrunk down to a smaller size, and then returned to original size with deterministic losses. This allows transmission of audio through narrow bandwidth data paths, and audio data to be stored in a much smaller memory space.

Audio compression and decompression algorithms are typically implemented on personal computers or embedded microprocessors. Although these implementations are cost-effective, these processors are designed for general-purpose applications and are therefore not optimised for the large repetitive mathematical operations needed in audio processing. Further, all samples and constants as well as the results of mathematical operations are, by definition, fixed to the bit width of the processor. As a result, there will typically be a tension between mathematical precision and the speed of compression or decompression of data. In a domain in which real-time performance is important, this trade off can lead to sub-optimal performance.

In comparison, Field Programmable Gate Arrays (FPGAs) do not have the limitations of bit length that characterises microprocessors. Without the limitations of bit widths and lengths the full resolution of each sample and constant and the full result of each mathematical operation can be used. By directly implementing the design in hardware,

using the Very high speed integrated circuit Hardware Description Language (VHDL), for example, the lengths of constants and mathematical operations may easily be varied, permitting direct comparisons to be made between the bit lengths and the accuracy of decoded audio data. In this way, various design tradeoffs can be made—biasing the design towards higher accuracy, for example, or alternatively towards higher performance or smaller area.

The work in this thesis has focussed specifically on the design and analysis of a MPEG (Moving Picture Experts Group) 1, Layer I audio decompression hardware unit as defined in the standard: ***ISO/IEC 11172-4:1993 Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s - Part 3 Audio*** [1].. Currently, research exists relating to the use of variable bit length decoders for MPEG decoding, but these are specifically related to variable sample bit lengths. In contrast, the main objective of the work in this thesis has been to determine the most efficient bit lengths for the various multiplication stages whilst still meeting the standard for MPEG compliance. To this end, a VHDL model of a MPEG 1 decoder has been built and its capabilities for decoding and decompression have been tested under various bit lengths.

The VHDL model was analysed in two stages. Firstly, a test waveform file (named `FL4.mp3`) from the ISO/IEC standard [2] was applied and compared to the output of a reference model written in the C computer programming language. Once the VHDL model was found to be compliant, the bit lengths of the various multiplication stages were then increased and decreased. The same test waveform was then applied to

determine if the model was still compliant at the new bit lengths of the multipliers. This was then repeated until the smallest bit lengths that would still allow compliance.

## 1.2 Research Scope

As mentioned in the overview, this work has focused specifically on the design and analysis of a hardware system supporting MPEG 1 Layer I, as defined in the standard ***ISO/IEC 11172-4:1993 Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s - Part 3 Audio*** [1]. The work was limited to MPEG 1 for two main reasons. Firstly, all later forms of MPEG audio compression have to be backward compatible with earlier versions, so this allows this work to be the basis for continued development by my employer, Semitech Innovations. More importantly, Semitech Innovation has an ongoing need for access to high quality audio compression technology for use in various telecommunications products. At the time this research was commenced there were no freely available implementations of MPEG audio compression available in VHDL form. All currently available implementations are in a precompiled form, or are of a proprietary closed source. As a result the work has also focussed on the design and development of a number of synthesisable and easily extensible MPEG hardware building blocks.

Note that this investigation has not covered the decoding of stereo, dual channel or joint stereo compressed data, nor will it include a CRC error check or any ability to decode the ancillary data. The parts of the VHDL model that would require changing to include the ability to decode stereo, dual channel and joint stereo would be the polyphase filter. This has been left for future enhancements to the research work.

## 1.3 Objectives

The objective of this investigation was to investigate the relationship between the numerical precision of the mathematical operations and overall error rate, in audio decompression systems and how these will affect both the error and the required hardware resources. The experiments were conducted on a MPEG 1 Layer I audio decoder system implemented in VHDL. The design was organised to allow the number of bits of mathematical operations, and therefore the numerical precision, to be easily increased and decreased for testing purposes.

To implement a MPEG 1 Layer I audio decoder system into VHDL, the number of processes and components required for decoding and decompressing a MPEG data stream first had to be investigated, in order to find the most efficient, in terms of speed and power, method of implementation. The model had to meet the accuracy and error limits defined in the ***ISO/IEC 11172-4:1993 Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 4 Compliance testing*** [2].

The VHDL model of the audio decoder was tested for both full and partial compliance [2] as well as to determine the minimum number of bits of resolution required for each multiplication stage while still meeting the compliance standards for MPEG 1 Layer I audio. This was performed for two cases: firstly where the number of bits of resolution for each of the four multiplication stages was varied independently, and then when they were varied in unison.

## 1.4 Methodology

The design and analysis work in this thesis has proceeded in four main stages:

1. A VHDL model of a MPEG 1 Layer I audio decoder was built using VHDL. The model was designed according to the ***ISO/IEC 11172-3:1993 Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 3: Audio*** standard [1]. The advantage of a VHDL model over, for example, a microprocessor or a Digital Signal Processor (DSP) implementation is its ability to be quickly modified without requiring a time consuming redesign.

2. A reference model built in the C programming language was sourced from the ISO/IEC standards site [3] and modified to suite the purposes of this research.

3. Initial tests were carried out on the VHDL model to determine whether the implementation was compliant to the standard ***ISO/IEC 11172-4:1993 Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 4 Compliance testing*** [2]. This was achieved using a test waveform from the standard (`FL4.mp3` from [1, 2]), which when decoded and decompressed by the C model gave 18,816 samples of digital audio data. The same input file applied to the VHDL model also resulted in 18,816 samples that could then be directly compared.

4. Finally, testing was undertaken to determine the minimum number of bits of resolution for each multiplication, while still meeting the standard for MPEG 1 Layer I audio full compliance, and partial compliance [2]. This was performed for two cases: where the number of bits of resolution for each multiplication was varied independently, and when they were varied in unison.

The final resolution tests (step 4, above) were carried out on the four principal multiplication stages in the decoding of a MPEG 1 Layer I compressed audio sample. The resolution (number of bits) of each multiply functional unit was decreased until the final output no longer fully met the MPEG 1 Layer I audio compliance standard [2]. The resolution was then further reduced until the final output no longer partially met the compliance standard. This process was then repeated, this time simultaneously decreasing the resolution of all the multiply units.

Only one of the test waveforms supplied with the standard (`FL4.mp3` from [1, 2]) was used in this work in order to limit the amount of data generated in testing, and to simplify implementation by restricting the design of the VHDL model to a MPEG 1 Layer I audio decoder. The particular waveform chosen is a constant amplitude sine wave with its frequency swept across the range of frequencies to be decoded. The compressed form of this waveform uses a bit-rate of 32kbit/s and a sampling frequency of 32kHz. This was decoded and decompressed by both the C-language and VHDL model to give 18,816 samples of digital audio data, as mentioned above.

## 1.5 Specific Outcomes

The thesis has resulted in the following deliverables:

- a VHDL implementation of a MPEG 1 Layer I decoder model where the length of the multiplier bits can be increased or decreased easily for the purposes of compliance and bit error testing;

- a version of the open source C model implementation distributed by ISO/IEC [3] that was modified to allow the resultant of each mathematical operation to be stored in a text file for comparison and the bit length of the multiply unit to be varied;
- test results leading to a better understanding of the relationship between bit length and accuracy of decoded compressed audio.

# Chapter 2. Data Compression and the MPEG Algorithm

Data Compression is the representation of information as accurately as possible using the least number of bits, i.e., representing a large amount of data with a smaller amount of data. This is useful because it decreases the amount storage space required for data, and this in turn leads to a reduction in the amount of time required to transmit data to another location. Data is compressed by a *compressor* to give compressed data and decompressed by a *decompressor* to give decompressed data.

## 2.1 The Theory of Data Compression

Coding theory is a topic that encompasses the disciplines of both mathematics and computer science in that it deals with the mathematical manipulation of data in an effort to reduce the error that occurs when the processed data is transmitted over a medium.

There are two main areas of coding theory: channel coding and source coding. Channel coding is more commonly referred to as forward error correction. In this case, information is processed in such a way as to reduce the probability that errors will be introduced into the data during transmission. Source coding is more commonly referred to as data compression, as information is processed so that a representation of the information is as accurate as possible using the least number of bits.

Data Compression generally starts by converting a continuously varying signal into a series of numbers with the use of an analogue to digital converter (ADC). These numbers are the uncompressed data. This uncompressed data is then passed through a series of mathematical processes to compress the data, as shown in the Figure 1.

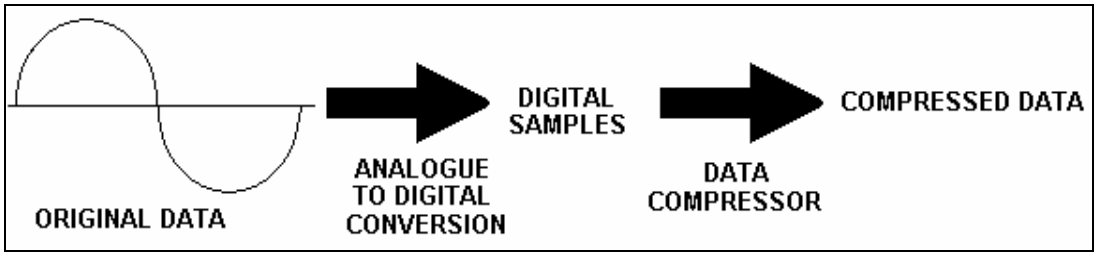


***Figure 1*** *Operation of an audio compressor*

Data Decompression passes the compressed data through the reverse series of mathematical processes to decompress the data. This decompressed data is then another series of numbers. Depending on the application, these numbers may finally be converted back into a continuously varying signal with the use of a digital to analogue converter (DAC), as shown in Figure 2.

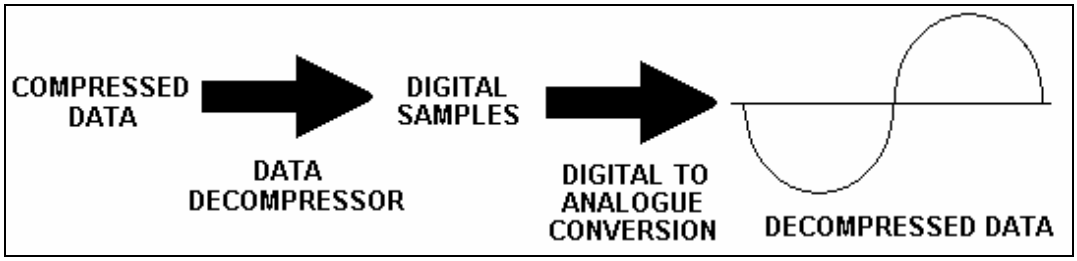


***Figure 2*** *Operation of an audio decompressor*

Data Compression may be separated into two types: *Lossless* and *Lossy*. In Lossless Data Compression all the information is retained in the compressed data after the

compression stage, thereby allowing the decompression stage to replicate an identical implementation of the data before compression [4]. The requirement that the decompression stage achieve a perfect replica of the data prior to compression leads to a reduction of the level of compression that can be achieved, typically to a ratio of around 2:1 (i.e., two units of uncompressed data lead to one unit of compressed data) [4]. As there is no difference between the decompressed data and the original source, lossless compression is used for application domains such as computer data and medical and scientific imaging.

In the case of Lossy Data Compression redundant, unnecessary or unimportant data, is removed in the compression stage. While this allows greater compression, the decompressed data is not an exact match to the original. The objective here is to avoid a noticeable degradation between the decompressed data and the original source. This form of compression is used where the exact data is not required, for example, in still images as well as audio and video transmission.

## 2.2 Audio and Speech Coding

Speech coding is the term used for the compression of a human voice. The various components of the human vocal cords and the glottis produce a speech waveform containing a fundamental or pitch frequency as well as other frequency components in the range 50Hz to 4kHz, that are then filtered by the nose and the throat forming a resonant frequency [5, 6]. Audio coding is the more general term used for the compression of audio that is not limited to voice, such as music and musical

instruments and film soundtracks. It puts an emphasis on the high quality representation and reproduction of the original varying analogue waveform.

As mentioned above, most forms of speech and/or audio coding require a continually varying analogue signal to first be converted into a digital form through the use of an analogue to digital converter (ADC). The analogue to digital conversion typically samples a continually varying analogue signal at regular period of time, and quantising the sampled value to a set of values representing the amplitude level. When a signal is quantised the amplitude of the signal is approximated to a smaller set of numbers by rounding the signal value to the closest value from the smaller set of numbers, thus reducing the information required to represent the amplitude of a signal, but at the same time ensuring information will be lost. By using finer quantisation the signal can be more accurately represented but this requires more information. The difference between the actual value of the signal and the value it is quantised to is referred to as the quantisation error [6]. The effect of this quantisation error is to add noise to the signal. This becomes apparent when the signal is reconstructed into a continually varying analogue signal.

The information required to represent the amplitude of a sample may be decreased by *companding*. Companding is a non-linear mechanism that reduces the gain of a signal after it reaches a certain threshold, either in the analogue domain, or after sampling and quantisation [6]. This means the value used to represent a large amplitude above the threshold is only slightly larger than one used to represent a small amplitude below the threshold. Companding is also referred to as non-uniform quantisation because a

change in the amplitude of the waveform does not always result in the same difference between the smaller set of numbers used to represent the waveform. Two examples of companding are A-law and Mu-law. A-law is used in Europe, while the Mu-law standard is more typically used in North America and Japan.

*Expanding* is the complement to Companding and works by reducing the gain of a signal after it falls past a certain threshold. Upon the reconstruction of the analogue signal the gain has to be reduced once it falls to a threshold so that the amplitudes of the signal are restored to the correct amount. Companding is also the basis of the Dolby noise reduction system [7], which uses companding to move a recorded audio signal into a higher frequency band away from low frequency noise.

The Nyquist Shannon Theorem, also more commonly referred to as Sampling Theorem, states that to represent a signal without error the sampling frequency must be at least twice the bandwidth of the signal [8]. That is, if the analogue signal is sampled at a frequency rate that is twice the maximum frequency component of the analogue signal then the signal may be reconstructed without error. Bandwidth Frequency limiting (sometimes referred to as pre-filtering or band-limiting), where the bandwidth of an analogue signal is limited to a smaller range through the use of an anti-aliasing filter, can be useful here because the corresponding sampling frequency needed to meet the Nyquist Shannon Theorem is reduced as a function of the bandwidth and thus the amount of information needed to represent the analogue signal is also reduced.

In sub-band coding, a continuously varying analogue signal is divided into a number of separate frequency components that can be independently compressed. In the case of speech coding, the lower frequency bands containing the fundamental frequency can use a larger number of bit per sample, while the higher frequency bands use a smaller number of bits per sample [6]. The division into a number of frequency bands is achieved through the use of a bank of band-pass filters. Upon reconstruction the different frequency bands are combined to form the original continuously varying analogue signal. An alternative filter mechanism uses the Fourier Transform to perform the frequency band separation [6].

Auditory masking occurs where a loud signal obscures the detection of another (e.g., a loud signal masking a quieter one) [6, 9]. The threshold at which the human auditory system can hear a signal, below which it is being obscured by a larger signal, is called the masking threshold. The masking threshold is also affected by the frequency and duration of both signals. This is the basis of Perceptual Coding—a data compression technique whereby information that is not distinguishable (by a human ear) is removed from the data [6, 9]. When the amplitude of a particular frequency component of an audio signal is below that of the threshold of human hearing, it is reasoned that this frequency component will not be heard and may be safely removed. Thus perceptual coding is a form of lossy compression that makes use of sub-band coding techniques.

Linear Predictive Coding (LPC) is a technique in which a number of the previous samples of a signal are used in a linear function to determine the likely future value of the signal. The difference between the predicted value and the actual value is then

stored [5, 10]. The number of samples used for the calculation of the future value is referred to as the *order* of the linear prediction coder. Unlike the previous techniques discussed above, which use the actual samples of continuously varying analogue signal, LPC uses the difference or error.

## 2.3 Types of Audio Compression Algorithms

Even from the early days of the telephone, audio compression is a topic that has received much attention. This section presents a small number of audio compression algorithms leading to a brief overview of the various MPEG standards.

Vocoder is an abbreviation of *Vo*ice En*coder* and was developed in 1939 by H.Dudley [11]. A Vocoder works by determining the carrier frequency of normal speech of a human voice, and then measuring the change in the frequency spectrum over time when the person is talking. These spectral changes are represented by a series of numbers, which are then transmitted. At the receiving end an oscillator creates the fundamental frequency that is then passed through a series of filters controlled (on/off) by the received data values. The outputs of these filters are summed to recreate the original frequency spectrum.

Pulse Coded Modulation (PCM) is the earliest form of an audio coder and was developed in 1948 [6, 12]. It comprises of an analogue to digital converter (ADC) sampling a continually varying analogue signal at a regular period thereby converting it into digital form and quantising the sampled value to a set of values representing the amplitude level. Differential Pulse Coded Modulation (DPCM) extends this basic idea

by taking the current sample of the continually varying analogue signal and using it to predict the value of the next sample. The difference between the predicted and the actual value of the sample is then quantised at a uniform rate [6]. In this way, the original signal can be represented with less information than PCM.

Adaptive Differential Pulse Coded Modulation (ADPCM), like DPCM takes the current sample of the continually varying analogue signal, and uses it to predict the value of the next sample. The difference between the predicted and the actual value of the sample is then quantised to form the modulation value. However, unlike DPCM, the quantisation of ADPCM is not uniform so that it is actually performing companding at the same time [6]. This means that the original continuous signal can be represented with even less information than DPCM.

Code Excited Linear Prediction (CELP) attempts to predict the speech value by passing a number of the previous samples through a filter thereby generating a number of pulses that are used by a speech synthesiser and compared to the actual speech sampled [10]. The error or difference between the predicted speech and the sampled speech is calculated and this error is transmitted [13].

MPEG audio compression is a lossy compression system that makes use of companding, subband and perceptual coding techniques to remove data that cannot be heard by the human auditory system. This is achieved at the encoding stage by using a bank of band-pass filters referred to as polyphase filters, along with a psychoacoustic model. The MPEG 1 and MPEG 2 audio compression standards were designed to be used for the compression of high quality audio. There are two main differences

between MPEG 1 and MPEG 2 audio compression. MPEG 1 is defined by the standard ISO/IEC 11172-3 [1] and uses sampling frequencies of 32kHz, 44.1kHz and 48kHz. MPEG 2 is defined by the standard ISO/IEC 13818-3 [14] and uses sampling frequencies of 16kHz, 22.05kHz and 24kHz. Further, MPEG2 allows for more than 2 channels of audio data.

MPEG 1 and MPEG 2 have three levels of complexity called Layer I, Layer II and Layer III. MPEG 1 and MPEG 2 audio compression exploit both sub-band coding and perceptual coding techniques. In Layer I and Layer II the digital representation of an analogue signal is divided into 32 sub-bands, each of the same bandwidth, and then passed through a psychoacoustical filter with 512 points for Layer I and 1024 points for Layer II. Layer III uses a hybrid form of sub-band coding that includes companding and that divides the digital representation of an analogue signal into 384 sub-bands.

MPEG 4 audio compression defined by the standard ISO/IEC 14496-3 [15] was designed for the compression of audio at a higher quality than the previous forms of MPEG audio compression. It is based on MPEG 2 Non Backwards Compatible (NBC), and consists of multiple audio compression systems such as Advanced Audio Coding (AAC), rather that being just a single implementation. Advanced Audio Coding makes use of larger number of filter banks and a prediction per subband whereby the error between the predicted value of the subband and the actual value is used to represent the signal.

## 2.4 MPEG Audio Compression

One of the advantages of audio compression is the ability to reduce the amount of memory or bandwidth required for the storage or transmission of digital encoded audio. A disadvantage is the complexity of the encoding and decoding processes, the amount of hardware required, as well as its energy and time. There is typically at least a linear relationship between the level of compression and the resources required.

One of the most widely used forms of audio compression is MPEG (Moving Picture Experts Group) 1 and 2 audio compression [16]. MPEG audio compression is actually one part of a three-part compression system used for the compression of moving video and images, although each part can operate independently of the others. One part is used for audio as mentioned above, one part for video imagery and another part for systems to combine the audio and video for storage and distribution. The key differences between MPEG 1 and MPEG 2 for audio is the addition of extra channels of stored sound and the option of lower sampling frequencies. MPEG 1 and 2 audio compression consists of three different types, or *layers*, referred to as Layer I, Layer II and Layer III. As the Layer number increases, so does the amount of compression, but also the complexity of the compression and decompression.

While the syntax of the coded bitstream is mandated, and the decoding process defined in terms of mathematical formulae in the standard, no implementations are specified (refer to 2.4.1). Thus, particular implementations may cater to specific needs and applications, so long as they pass compliance testing for accuracy [2]. In this way,

future encoder/decoder designs may take advantage of improvements in technology as they become available.

A block diagram of a generic MPEG audio encoder is shown in Figure 3. The analogue input signal is sampled into a stream of pulse code modulated data that is passed to a bank of polyphase band-pass filters and simultaneously through a psychoacoustic model. As mentioned previously, there are 32 band-pass filters for Layer I and Layer II (and 384 for Layer III), so here the analogue signal is divided into 32 frequency bands. The psychoacoustic model is used to determine the number of quantisation levels required to make the quantisation noise inaudible in each frequency band [17]. There are two types of psychoacoustic models defined in the standard [1]. Model 1 is for use for Layer I and Layer II. The alternative Model 2 can be used for all three Layers. Model 1 is less complex due to compromises made to simplify the calculations. The psychoacoustic models also make use of Fourier transforms to transform the audio from the time domain to the frequency bands. The quantisation levels and the subbands of frequency are then bit allocated. The bit allocation determines the number of bits that will be used to represent each frequency sub-band based on the number of quantisation levels determined by the psychoacoustic model. This information is formatted into a bit-stream containing the bit allocations and scalefactors for each of the 32 subbands.

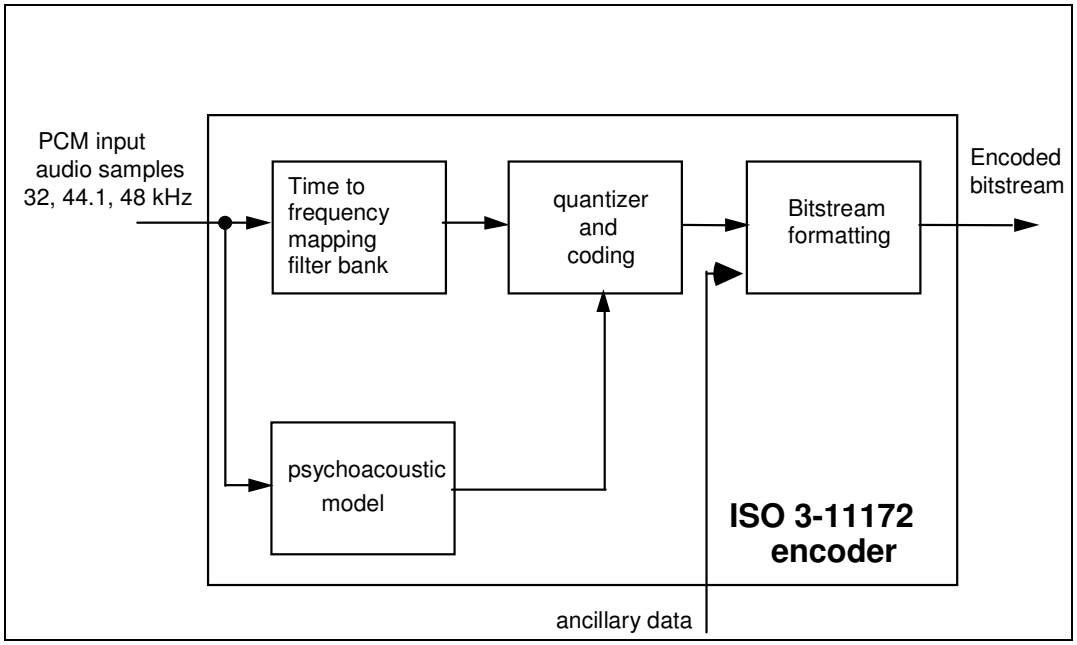


***Figure 3*** *A block diagram of a MPEG audio encoder*

The MPEG audio decoding process shown in Figure 4 separates the encoded bit-stream into its components. The header of the bit-stream is used to determine the number of bits per sub-band sample plus the scale factors. The 32 subbands per sample are then rescaled [17] and the sub-band samples transformed from the frequency domain to the time domain through the use of a filter. The filter output is a stream of pulse code modulated data that is converted by a digital to analogue converter back into the analogue domain.

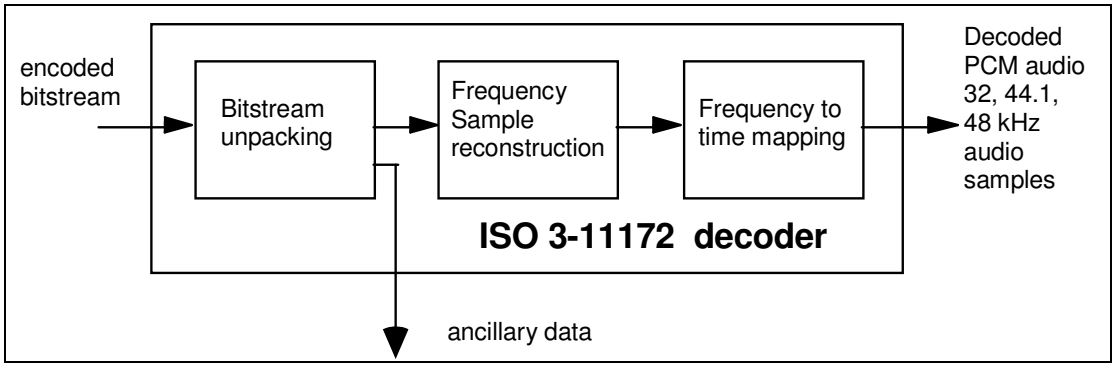


***Figure 4*** *A block diagram of a MPEG audio decoder*

One area of MPEG audio compression that requires significant resources is the polyphase synthesis sub-band filter used in both encoding and decoding. The purpose of this filter is to remove those frequency components that are least likely to be heard. Research relating to the design of the polyphase synthesis sub-band filter in the decoding of MPEG compressed audio data takes the objective of trying to reduce mathematical operations required through the use of mathematic equivalence formulas.

## 2.4.1 MPEG Implementations

A reference implementation of MPEG audio encoding and decoding was released in 1996 by the ISO MPEG Audio Subgroup Software Simulation Group under the title ***ISO dist10 Source Code Package*** [3]. This program is written in the C computer programming language to show the most pure (functional) implementation of MPEG compression and decompression. This reference program is highly inefficient, and will not decode MPEG compressed audio in real time, even on a current personal computer.

Early implementations were all software. These still have a major advantage that they are cheap to develop and make use of low cost commercially available general purpose embedded processors. However the major disadvantage is the fixed bit length of the general purpose embedded processor. Mathematical operations are forced to use a word-length that insures they do not overflow. Another disadvantages may be the lack of specialised support in the general purpose embedded processor hardware and the propriety nature of most products and implementations. The propriety of most implementations means that a large licensing fee is required for use in a product or to obtain the source code. In the filter design of a MPEG 1 Layer I audio decoder, considerable processing time is spent performing multiply and accumulate operations as part of the polyphase synthesis sub-band filter modelling. Unlike DSPs, which contain specific hardware for combining and performing multiply and accumulate operations, a general purpose embedded processor typical requires several instructions to perform the same operation. DSPs have a great advantage in decreasing the processing time spent performing multiply and accumulate operations by providing specific hardware. On the other hand, this leads to a higher dollar value for the DSP in comparison to a general purpose embedded processor. Almost all research activities related to the development of VHDL versions of MPEG decoders are highly confidential due to the valuable nature of that research. Prior research into the use of variable bit length decoding in VHDL implementation models of MPEG decoders has mainly focused on MPEG 2 video implementations, rather than MPEG 1 audio [18, 19]. The research in this thesis explores the use of variable bit lengths to find an equivalent pre calculated value from a look up table.

## 2.5 The Historical Background of MPEG

The Moving Picture Experts Group (MPEG) was formed by the International Standards Organisation (ISO) in 1988 to create standards for the coded representation of moving pictures, and the associated audio, for storage on digital media. The MPEG audio standards were submitted in November of 1991, and adopted as a standard at the end of 1992 [1, 17].

The standards for MPEG compression exist in three parts with Part 3 dealing with the compression and decompression of audio. The MPEG audio standards are entitled ***Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 3: Audio*** and assigned the standard number 11172-3 [1], (where the 1.5 Mbit/s refers to the combined bit rate of video and audio). In May 1994, the MPEG standards for audio were updated for lower sampling frequencies and expanded to cover multi-channel (more than 2 channels) surround-sound and given the standard number 138183-3, while keeping the same title [14].

The MPEG 1 and 2 audio compression standards [1] have become very popular and have become the universal standard for applications such as consumer electronics, professional audio processing, telecommunications and broadcasting [16]. This is largely because that MPEG /audio compression is the first international standard for the digital compression of high-fidelity audio [17]. Today MPEG 1 and 2 audio compression technology is used in a wide range of consumer items from portable music players, DVDs, digital television and radio broadcasting, telecommunications as well as for the exchange of music on the Internet. It is popular across these diverse fields

because it is able to compress compact disc (CD) quality audio to a bit rate of 128 kbit/s from the 706 kbit/s of uncompressed CD quality audio [20] (i.e. a compression ratio of around 6:1).

Since its initial implementations, a number of improvements to MPEG 1 and 2 audio encoders and decoders have been published. Most of these result have been aimed at improving the polyphase synthesis sub-band filter including a fast Discrete Cosine Transform [21], an improved fast Discrete Cosine Transform [22] and matrixing operations [20].

## 2.6 The Current Uses of MPEG Data Compression

The original purpose for the publication of the MPEG standard was to allow the compression of video and audio data so that it can use readily available digital storage media. As identified previously, MPEG supports compression ratios of up to 6:1 - i.e., six times the amount of audio information can be stored at CD quality compared to the uncompressed case [20]. This is a dramatic reduction in the size of storage required, which leads to cheaper costs.

The Compact Disc (CD) brought about a revolution in cheap digital storage media. Video and audio compression were combined with CD media to form the Video Compact Disc (VCD). This format uses MPEG 1 compression for video and audio to compress a 74 minutes of VHS (Video Home System) quality video and CD quality sound onto a standard CD. Thus two compact discs, or one double-sided disc, could

store a full-length feature film. This has lead to the use of MPEG 2 compression for video and audio in the DVD (Digital Versatile Disc). While these discs are similar in size to a CD, they have a different physical method for storing data and therefore a much higher data storage capacity. In this case, MPEG compression is an option for the audio component and competes with other digital compression technologies such as Dolby Digital AC-3 (Adaptive Transform Coder 3), and DTS (Digital Theatre Sound).

MPEG video and audio compression is being used in various free to air, satellite and cable television broadcasting services for digital television. Compared to existing analogue systems the use of compression on digital transmission greatly reduces the bandwidth required per channel. The extra bandwidth available allows either more channels within the same RF bandwidth or additional information to be transmitted. This extra information may take the form of sports scores, share indexes etc, alternate camera angles, or alternate soundtracks and commentaries, for example. Alternatively the extra bandwidth may be used for completely new uses such as radio communications or wireless Internet access. Digital Audio Broadcasting (DAB) based on the Eureka 147 DAB also uses MPEG 1 Layer II for audio to broadcast digital radio [23]. It is currently in trials in Australia and is being used in regular service in Europe and Canada. Once again the advantage of digital transmissions is that with the use of compression on digital information, the radio bandwidth required per channel is greatly reduced.

The large growth of music distribution (both legal and otherwise) via the Internet has been fuelled by the use of Layer III MPEG audio compression. This is because the

time needed to download an audio track compressed using MPEG compression can be as little as $1/6^{th}$ of the time required for an uncompressed track, making it easier to move music files around the Internet. The key advantage for record companies is distribution cost. Whereas the more normal distribution model requires a CD to be manufactured and packaged, warehoused and transported before being stocked on a shelf prior to eventual purchase, an Internet distribution only requires servers and an Internet connection, making the Internet distribution model extremely cost effective.

The large amount of MPEG compressed audio on the Internet, and the ever decreasing cost of digital media, has led to portable players and car audio players that can decompress the encoded MPEG audio in real time. Consumers may now carry a larger selection of music with them than would be possible using existing portable tape or CD. These new portable MPEG audio devices combined with the distribution of music using the Internet has led the MPEG standards to become increasingly popular.

## 2.7 Theoretical versus Practical Design Constraints

An important part of the implementation of a MPEG audio decoder is the polyphase synthesis sub-band filter. This comprises two mathematical equations, one for Matrixing and the other for Windowing. In their original form, the mathematical equations used in the polyphase synthesis sub-band filter that is used in the decompression of compressed MPEG audio, is time and resource intensive. It has been reported that the sub-band filter synthesis operation represented 40% of the overall decoding time [20], and 70% of the mathematical operations needed for the decoder [24]. Significant speed and memory improvements have been made to the synthesis

filter by making use of an improved fast Discrete Cosine Transform [22]. This direct factorisation has been shown to lead to an improvement in the order of six times the speed of the standard Discrete Cosine Transform (DCT) through the reordering of the mathematical processes and discarding the mathematical operations taking place in the imaginary domain. The calculations required for the evaluation of a DCT can be further optimised via the use of indirect computation algorithms leading to another reduction of around 25% [25].

Further optimisation has been achieved by a fast DCT that halved the number of multiplication operations [21]. Efficiency was then improved again in [20] by optimising the matrixing operations such that the number of multiply instructions could be reduced from 2048 to just 80 multiplications plus 209 additions. The clear advantage here is that additions are less hardware intensive than multiplications on standard microprocessors.

This has led to further research into reducing the number of multiplications at the expense of an increase in the number of additions and the removal of floating point multiplication by the use of integer multiplications [26]. Integer multiplications are preferable because they are typically much smaller than the corresponding floating point units. Mathematical factorisations can be applied such that only the real components of frequency remain. This simplifies the overall computations [27] but increases the number of multiplications and additions required. In [28], factorisation was used with a new fast algorithm that kept the number of multiplications and additions the same but expressed them in a more easily computed form that decreased

computation time. In that work, a 16 bit integer decoder was built that reduced the number of multiplications in a fast Discrete Cosine Transform by 28% but, importantly, changed these multiplications from floating point to integer [29].

The word length has been investigated in [24] in an effort to minimise the hardware area required, without introducing additional noise. This resulted in a 21-bit word length for the multiplier, and a 25-bit accumulator. However this method was aimed at a microprocessor implementation and therefore was forced to use the same multiplier and accumulator for both the matrixing and the windowing operations of the polyphase synthesis sub-band filter. Also, the word lengths were forced to be an integer multiple of the basic word length.

The use of distributed arithmetic to reduce the amount of area required is shown in [30]. Distributed arithmetic uses less hardware than conventional architectures by replacing multiplier units with look up tables and accumulators. A more efficient windowing system that reduced the number of multiplications from 512 to 285, while increasing the number of additions from 480 to 806 was described in [31].

## 2.8 Current Approaches to MPEG Compression

As discussed in Section 2.7, it can be seen that there are four main approaches currently in use for implementing the matrixing operation of the polyphase synthesis sub-band filter,

1. the “brute force” used in way shown in the standard [1];
2. Lee’s method using the fast Discrete Cosine Transform [21];

3. Chen's method using an algorithm for the fast Discrete Cosine Transform [22];

4. Konstainides method of removing the imaginary components [20].

All of these methods are intrinsically serial in nature and employ a fixed word length mandated by their microprocessor implementation. In comparison, a programmable logic implementation can access whatever parallelism is inherent in the algorithm and at the same time exploit a word length that is determined to be the best compromise between area, performance and quality.

## 2.9 Description of an MPEG Frame

A sample of compressed MPEG 1 Layer I audio data comprises a series of frames. As can be seen in Figure 5, each frame of compressed MPEG 1 Layer I audio data has four parts: the header, the error check, the compressed audio data and the ancillary data. While the header and the compressed audio data will always be present, the error check and the ancillary data are optional.

The header contains 32 bits of basic information needed to configure the MPEG 1 Layer I decoder and to determine how the compressed audio data in the third part of the MPEG 1 Layer I audio frame needs to be to decompressed. The operation of the header is described in the following section.

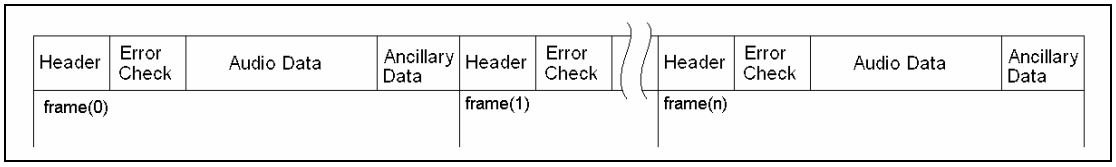


***Figure 5*** *Structure of an MPEG 1 Layer I audio frame*

The error check is an optional 16 bits of information making up a cyclic redundancy check (CRC) for the audio frame data. This CRC is calculated for the data in the header as well as for the compressed audio data. When the audio decoder receives this error check, it calculates its own CRC and if they are the same it is assumed that that overall frame was received correctly. As the CRC error check is optional, the decoder checks a header flag (active low) to determine if it is present. If not, the frame contains only the header, the compressed audio data and possibly ancillary data. Note that the ancillary data is not part of the ISO/IEC 11172-3 standard [1]. It is always an optional component that has to be defined by the user. As such, it was not used in the work reported in this thesis.

### 2.9.1 The MPEG Frame Header

The first part of the frame of compressed audio data is the header. The header is 32 bits in size and contains thirteen pieces of basic information needed to configure the decoder. An example of the header of a MPEG 1 Layer 1 audio frame is shown below in Figure 6. These thirteen parts are described as follows:

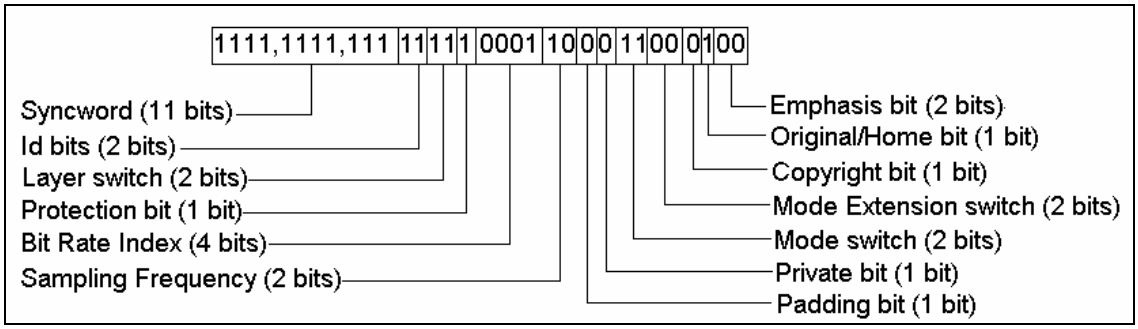


***Figure 6*** *Header of an MPEG 1 Layer I audio frame*

***Syncword (11 bits)***

This field represents a series of 11 high bits, (i.e., 111, 1111, 1111) and is used to determine the start of the frame, signalling the start of the decode process.

***ID bits (2 bits)***

The ID (identifier) bits are used to identify a MPEG frame of data. In the first MPEG standard (11172-3) [1], the ID was a single bit and the sync-word was 12 bits, however in the later standard (13818-3) [14], the ID was specified as a pair of bits and the syncword was decreased one bit to 11 bits to compensate. This work is based on the 13818-3 standard [14], in order allow for future expansion. The MPEG 1 audio decoder will wait for the syncword of the next frame before starting to decode the next header. The ID switch codes are shown in Table 1. As can be seen in the example shown in Figure 6, valid ID bits were found because they were both set high (i.e., 11) meaning the compressed data in the frame is MPEG 1 formatted.

| ID Switch | MPEG Version |
|---|---|
| "11" | MPEG 1 as defined by standard 11172-3 [1] |
| "10" | MPEG 2 as defined by standard 13818-3 [14] |
| "01" | Reserved |
| "00" | MPEG 2.5 a non standard approved extension to the original standards [32] |

***Table 1*** *ID bits and MPEG Versions*

### *Layer switch (2 bits)*

The layer switch follows the ID bits and is used to determine the level of compression that has been used for the compressed audio data in the third part of the audio frame. There are three layers of audio compression in MPEG 1 audio. Layer I has the least compression, but is also the least complex to decode and decompress. Layer III is the most complex to decode and decompress, but offers the most compression. As to be expected, layer II offers an intermediate level of complexity and compression. The Layer switch codes are shown in Table 2. As can be seen in the example shown in Figure 6, the Layer switch is both bits set high (i.e., 11), which will set the MPEG 1 audio decoder to decode Layer I compressed audio data.

| Layer switch | Layer |
|---|---|
| "11" | Layer I |
| "10" | Layer II |
| "01" | Layer III |
| "00" | reserved |

***Table 2*** Layer switch codes

### *Protection bit (1 bit)*

The protection bit is used to inform the decoder whether the MPEG 1 frame of compressed audio data employs data correction. If set low (i.e., 0) data correction will be applied. If data correction is to be used, a cyclic redundancy check will be

calculated for both the header and the compressed data and compared with the CRC part of the frame. If the CRC comparison passes, the MPEG 1 Layer I frame is taken to be correct and the rest of the frame's data is decoded as normal. If a difference is detected, either within the error check or the rest of the frame, the decoder usually attempts to use the data from the most recent frame where the error check did match. The decoder may also try to request either a re-read from memory, or a retransmission in the hope that it will be error free. Alternatively, the decoder may try to apply error correction methods (although this is not covered in the ISO/IEC 11172-3 standard [1]).

If no data correction is to be used, the second part of the frame containing the error check will not be present. That is, the audio data frame will contain only the header and the compressed audio data[1]. The example of Figure 6 uses no data correction, thus the protection bit is set high.

***Bit rate index (4 bits)***

The bit rate index is a four-bit number used to represent the overall bit rate of data in the MPEG frame. It is used in conjunction with the layer switch as an index for the look up table shown below in Table 3. If the four bits are all low (i.e., 0000), then the decoder is said to use a 'free' format and the bit rate can take any value. Otherwise the bit rate takes on the table value. In the example frame shown previously in Figure ***6***, the bit rate index is set to 0001, so (from Table 3) the bit rate will be 32 kbits/second.

[1] As noted above, no ancillary data is present in this work.

| bit_rate_index | Layer I bit rate | Layer II bit rate | Layer III bit rate |
|---|---|---|---|
| '0000' | Free format | Free format | Free format |
| '0001' | 32 kbit/s | 32 kbit/s | 32 kbit/s |
| '0010' | 64 kbit/s | 48 kbit/s | 40 kbit/s |
| '0011' | 96 kbit/s | 56 kbit/s | 48 kbit/s |
| '0100' | 128 kbit/s | 64 kbit/s | 56 kbit/s |
| '0101' | 160 kbit/s | 80 kbit/s | 64 kbit/s |
| '0110' | 192 kbit/s | 96 kbit/s | 80 kbit/s |
| '0111' | 224 kbit/s | 112 kbit/s | 96 kbit/s |
| '1000' | 256 kbit/s | 128 kbit/s | 112 kbit/s |
| '1001' | 288 kbit/s | 160 kbit/s | 128 kbit/s |
| '1010' | 320 kbit/s | 192 kbit/s | 160 kbit/s |
| '1011' | 352 kbit/s | 224 kbit/s | 192 kbit/s |
| '1100' | 384 kbit/s | 256 kbit/s | 224 kbit/s |
| '1101' | 416 kbit/s | 320 kbit/s | 256 kbit/s |
| '1110' | 448 kbit/s | 384 kbit/s | 320 kbit/s |
| '1111' | invalid | invalid | Invalid |

***Table 3*** *Bit rate index switch table*

***Sampling frequency (2 bits)***

The sampling frequency switch is a two-bit number representing the sampling frequency used for converting the audio into a digital pulse code modulated waveform, as shown in Table 4. The example in Figure 6 is showing a sampling frequency of 32 kHz.

***Padding bit (1 bit),***

The padding bit is used to determine whether an extra slot of compressed audio will be used when the sampling frequency is set 44.1 kHz, so that the correct bit rate is

maintained. This will be explained in more detail in **Section 2.9.3**. The example shown in Figure 6 does not use padding and thus the padding bit is set low.

| Sampling frequency switch | Sampling frequency |
|---|---|
| '00' | 44.1 kHz |
| '01' | 48 kHz |
| '10' | 32 kHz |
| '11' | reserved |

***Table 4*** *Sampling Frequency switch table*

***Private bit (1 bit),***

The private bit is no longer used by the ISO standard and is only keep for backward compatibility with older decoders.

***Mode switch (2 bits) and mode extension switch (2 bits)***

The mode switch is used to determine whether the audio data has been encoded in one of four modes as follows (Table 5):

- **single channel** where one channel of data is encoded, and when decoded the same data is used for both the right and left channels to give monophonic sound;
- **stereo channel** where two separate single channels (one each for the left and right channels) of stereo data are encoded and when decoded form a stereo pair giving stereophonic sound;
- **dual channel** where two separate single channels (one for the first channel and one for the second channel) of non-stereo data are encoded and when decoded one channel is used for the left channel and the other for the right channel,

forming two distinct and separate channels and allowing a user to use a balance control to switch totally to the left, or right channel;

- **joint stereo** where the low frequencies common to the left and right channels of a stereo pair channels are encoded and stored once (in the left channel), while the higher frequencies that are not common to the left and right channels of a stereo pair channels are encoded and stored separately in two different channels (one for the left channel and one for the right channel, but due to the fact there are no low frequency components for the right channel, less room is required). When decoded, the left channel is used for both the left channel and the low frequencies and the data encoded for the right channel is used for the right channel higher frequencies to form a stereo pair giving stereophonic sound.

| Mode switch | mode |
|---|---|
| '00' | stereo |
| '01' | joint_stereo (intensity_stereo and/or ms_stereo) |
| '10' | dual_channel |
| '11' | single_channel |

***Table 5*** *Mode switch table*

In the example frame shown in Figure 6, both bits of the mode switch are set high (i.e., 11) indicating (from Table 5) that the audio frame is in single channel mode with no mode extension.

In joint stereo mode the ***mode extension*** switch bits are also valid. The mode extension switch (Table 6) is a two-bit number used to determine the number of sub-bands that

will be common to both channels in the joint stereo mode. These sub-bands are referred to as “bound” meaning that the number of subband samples that are common to both the left and right channels.

| Mode extension switch | Mode extension |
|---|---|
| '00' | sub-bands 4-31 in intensity_stereo, bound==4 |
| '01' | sub-bands 8-31 in intensity_stereo, bound==8 |
| '10' | sub-bands 12-31 in intensity_stereo, bound==12 |
| '11' | sub-bands 16-31 in intensity_stereo, bound==16 |

***Table 6*** *Mode Extension switch table*

***Copyright bit (1 bit)***

The copyright bit is used to specify whether the compressed audio data is copyrighted. If the copyright bit is set (high), the compressed audio data has copyright protection and home entertainment devices that support copyright protection will not allow a copy of the audio data to be made. The example shown in Figure 6 has the copyright bit is set low meaning that there is no copyright restrictions on the compressed audio data.

***Original/home switch (1 bit)***

The original/home bit determines whether the audio data is the original data or a copy. Home entertainment devices that support copyright protection will not allow a copy of the audio data to be made, regardless of the copyright bit. In this way, a copy may be made of the original but not of any copy. In Figure 6, the original/home bit is set high, meaning that no copyright restrictions exist on the compressed audio data.

| Emphasis switch | Emphasis |
|---|---|
| '00' | no emphasis |
| '01' | 50/15 μSec. emphasis |
| '10' | reserved |
| '11' | CCITT J.17 |

***Table 7*** *Emphasis switch table*

***Emphasis switch (2 bits).***

Finally, the emphasis field is a two-bit switch that determines the type of de-emphasis to be used on the samples of audio data. The de-emphasis process involves the application of filtering to the audio signal in order to reverse a prior emphasis that results in linear distortion. The options types are shown in Table 7. No emphasis is applied on the compressed audio data with the example shown in Figure 6.

## 2.9.2 The Error Check of a MPEG Frame

The second part of the MPEG audio frame is the error check. This is optional depending on the setting of the protection bit. If the protection bit in the header is set enabled, then a 16 bit cyclic redundancy check word is inserted in the bit-stream after the header. This 16-bit check word is used to check if there are errors in the header (starting with bit_rate_index and ending with emphasis), along with a set number of the audio bits. The number of audio data bits to be included is determined by "NUMBER OF PROTECTED AUDIO_DATA BITS" of the MPEG standard (Table 8) [14].

| Layer | bit allocation table number. | number of bits - single channel mode | number of bits - other modes |
|---|---|---|---|
| I | | 128 | 256 |
| II | 3-B.2a | 142 | 284 |
| II | 3-B.2b | 154 | 308 |
| II | 3-B.2c | 42 | 84 |
| II | 3-B.2d | 62 | 124 |
| III | - | 136 | 256 |

***Table 8*** *Number of bits per channel tables*

The data from the header plus the audio bits are used to calculate a cyclic redundancy check work internally using the "CRC-16" generator polynomial:

$$G(x) = X16 + X15 + X2 + 1 \quad (1)$$

as illustrated by the circuit shown in Figure 7, where X16 is the $16^{th}$ bit, X15 is the 15th bit and X2 is the 2nd bit of the generated polynomial G(x). Figure 7 shows a feedback shift register where the boxes T0 to T15 represent the bits of the CRC register. The arrow before T0 is the 1 from the CRC polynomial and the arrows between T1 and T2 and T14 and T15 being the X2 and X15 bits of the polynomial. The arrow out of T15 to the addition is X16. The output is a word compared with the error check word in the bit-stream. If these are not identical a transmission error is assumed to have occurred in the MPEG frame. Repetition of the previous frame is recommended in case of an error.

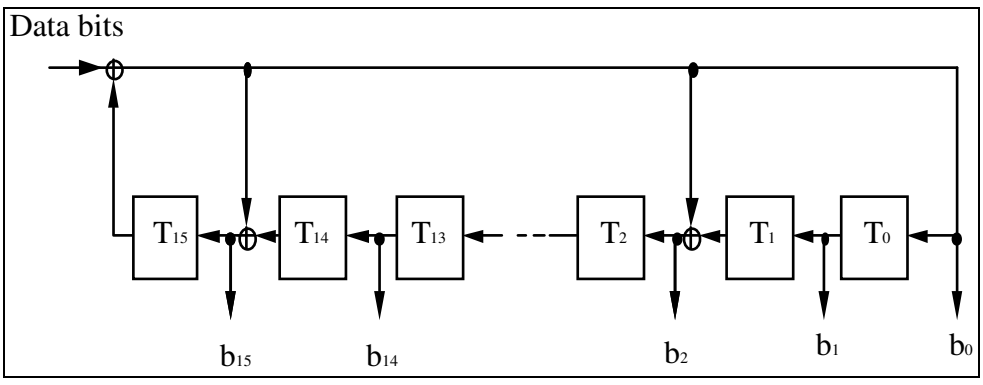


***Figure 7*** *Flowchart of Error Correction*

### 2.9.3 The Compressed Audio Data of a MPEG Frame

Using the information from the header, the MPEG 1 Layer I audio decoder is configured to decode this field into uncompressed audio samples. The process for decoding the compressed MPEG 1 Layer I audio data is shown in the flowchart shown in Figure 8.

The compressed audio part of the MPEG 1 Layer I audio frame can be considered to be divided into slots. The number of slots (N) will indicate the distance to the next sync word (this does not include any data from the ancillary data part of the audio frame). The number of slots before the next sync words can be calculated using the formulas:

Layer I $$N = 12 \times (bit\_rate / sampling\_frequency) \quad (2)$$

Other Layers $$N = 144 \times (bit\_rate / sampling\_frequency) \quad (3)$$

Most of the time the result of either (2) or (3) will give an integer. If the sampling frequency is set 44.1 kHz, N will not be an integer and it will need to be truncated and "padding" applied. Padding allows the number of slots to vary between N and N + 1. This is determined by the value of the padding bit in the header part of the MPEG 1 Layer I audio frame. If padding is enabled (i.e., 1), the number of slots will be N + 1. The only exception to this is if the bit rate index in the header part of the audio frame is set to free format, where there is no specified bit rate. In this case the value of N will have to be calculated from the distance between consecutive *syncwords* and the value of the padding bit.

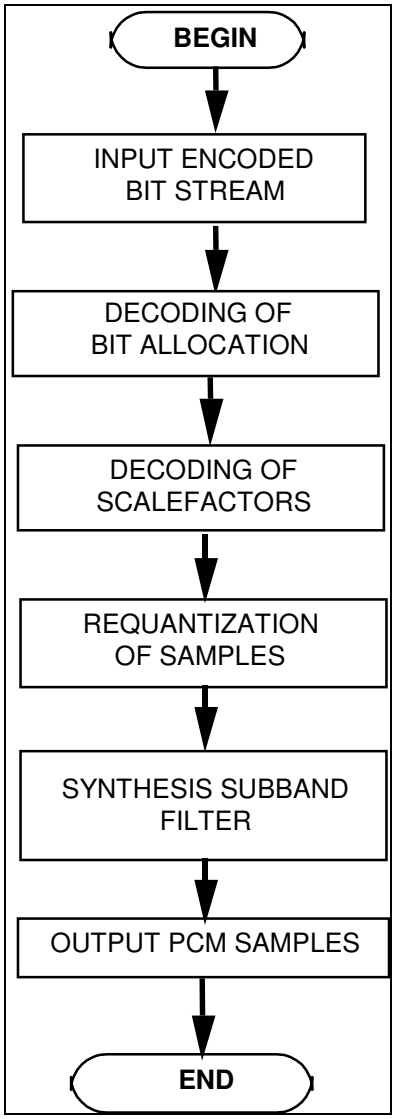


***Figure 8*** *Basic flowchart of Layer I & II decoding*

In the example shown previously in Figure 6, the Layer switch is set to Layer I, the bit rate index is to 32 kbit/second and the sampling rate set to 32 kHz. From (2), the number of slots of compressed audio data in this case will be 12, as follows:

$$N = 12 \times (bit\_rate / sampling\_frequency) = 12 \times (^{32(\text{kbit/s})}\!/_{32\ (\text{kHz})}) = 12 \times 1 = 12$$

### 2.9.4 The Bit Allocation Codes

The slots of compressed data from a MPEG 1 Layer I audio frame may contain three different types of data:

1. the bit allocation codes;
2. the scale factors;
3. the compressed sub-band samples of audio data.

For the audio frame example from Figure 6, the compressed audio data is single channel encoded. This means that 32 four-bit allocation codes are to be read per slot, as shown in the second process of Figure 8. These allocation codes are used as an index for a look up table that gives a value for the number of bits that will be read for each compressed sub-band sample of audio data (Table 9).

| Allocation code | Number of bits |
|---|---|
| '0000' | 0 |
| '0001' | 2 |
| '0010' | 3 |
| '0011' | 4 |
| '0100' | 5 |
| '0101' | 6 |
| '0110' | 7 |
| '0111' | 8 |
| '1000' | 9 |
| '1001' | 10 |
| '1010' | 11 |
| '1011' | 12 |
| '1100' | 13 |
| '1101' | 14 |
| '1110' | 15 |
| '1111' | invalid |

***Table 9*** *Number of Allocation bits look up table*

## 2.9.5 The Scalefactor Index

The second piece of information read from a slot of the compressed audio data part of the MPEG 1 Layer I audio frame is the scalefactor codes. For each of the allocation codes that are not equal to zero (i.e., 0000), a corresponding six bit number called the *scalefactor index* is read per slot, (third process of Figure 8). This scalefactor is then used as an index for a look up table that returns the scalefactor that is then used to multiply ("scale") the compressed sub-band sample of audio data. The scalefactor indices and the corresponding scale factor are shown in Table 10.

| Scale factor index | scale factor | Scale factor index | scale factor | Scale factor index | scale factor |
|---|---|---|---|---|---|
| 0 | 2.00000000000000 | 21 | 0.01562500000000 | 42 | 0.00012207031250 |
| 1 | 1.58740105196820 | 22 | 0.01240157071850 | 43 | 0.00009688727124 |
| 2 | 1.25992104989487 | 23 | 0.00984313320230 | 44 | 0.00007689947814 |
| 3 | 1.00000000000000 | 24 | 0.00781250000000 | 45 | 0.00006103515625 |
| 4 | 0.79370052598410 | 25 | 0.00620078535925 | 46 | 0.00004844363562 |
| 5 | 0.62996052494744 | 26 | 0.00492156660115 | 47 | 0.00003844973907 |
| 6 | 0.50000000000000 | 27 | 0.00390625000000 | 48 | 0.00003051757813 |
| 7 | 0.39685026299205 | 28 | 0.00310039267963 | 49 | 0.00002422181781 |
| 8 | 0.31498026247372 | 29 | 0.00246078330058 | 50 | 0.00001922486954 |
| 9 | 0.25000000000000 | 30 | 0.00195312500000 | 51 | 0.00001525878906 |
| 10 | 0.19842513149602 | 31 | 0.00155019633981 | 52 | 0.00001211090890 |
| 11 | 0.15749013123686 | 32 | 0.00123039165029 | 53 | 0.00000961243477 |
| 12 | 0.12500000000000 | 33 | 0.00097656250000 | 54 | 0.00000762939453 |
| 13 | 0.09921256574801 | 34 | 0.00077509816991 | 55 | 0.00000605545445 |
| 14 | 0.07874506561843 | 35 | 0.00061519582514 | 56 | 0.00000480621738 |
| 15 | 0.06250000000000 | 36 | 0.00048828125000 | 57 | 0.00000381469727 |
| 16 | 0.04960628287401 | 37 | 0.00038754908495 | 58 | 0.00000302772723 |
| 17 | 0.03937253280921 | 38 | 0.00030759791257 | 59 | 0.00000240310869 |
| 18 | 0.03125000000000 | 39 | 0.00024414062500 | 60 | 0.00000190734863 |
| 19 | 0.02480314143700 | 40 | 0.00019377454248 | 61 | 0.00000151386361 |
| 20 | 0.01968626640461 | 41 | 0.00015379895629 | 62 | 0.00000120155435 |

***Table 10*** *Scalefactor look up table*

### 2.9.6 The Compressed Audio Sub-band Sample

The third piece of information read from a slot of the compressed audio data part of the audio frame is the compressed sub-band sample. For each of the allocation codes that are not equal to zero (i.e., 0000), the number of bits determined by the bit allocation is read for each compressed sub-band sample. These groups of two to fifteen bits will make up one compressed audio sample sub-band. This is repeated 12 times for each of the 32 bit allocation values read.

## 2.10 The Requantisation of Compressed Audio Sub-band Samples

The compressed audio sample sub-bands now have to be requantised, as shown in the fourth process of Figure 8. Each time that the data bits for one sub-band sample of compressed audio data has been read, the first bit has to be inverted. This will give a two's complement fractional format binary number, where the most significant bit represents the value -1, and each of the following bits will represent a half of the digit before. For example, most significant bit is -1, the next significant bit is 0.5, the next 0.25 ….etc.

The compressed sample is restored to a fractional number using the function:

$$s'' = \frac{2^{nb}}{2^{nb}-1} \times (s''' + 2^{(-nb+1)}) \qquad (4)$$

where *s”* is the fractional number, *nb* is the number of bits allocated to each sample in the sub-band and *s’’’* is the unrestored two's complement fractional format number that was read and inverted in the previous step. It can be seen in (4), that after an unrestored two's complement fractional format binary number has been added to the constant

$2^{(-nb+1)}$, it is multiplied by the constant $2^{nb}/2^{(nb-1)}$, resulting in a restored two's complement fractional format binary number. This restored sub-band is rescaled by multiplying by the scalefactor determined previously (see Table 10). After the multiplication shown in (5) the sub-band sample is said to be a *rescaled* sub-band sample.

$$s' = scalefactor \times s'' \tag{5}$$

At this point a compressed audio sample sub-band is said to be *requantised*. The requantised compressed audio sample sub-band has to be passed through the polyphase synthesis sub-band filter, as shown in the fifth process of Figure 8, in order to calculate 32 decompressed audio samples.

## 2.11 The Polyphase Synthesis Sub-band Filter

Once 32 requantised compressed audio sample sub-bands have been calculated the Polyphase Synthesis Sub-band Filter is used to calculate 32 consecutive decompressed audio samples. The process for the polyphase synthesis sub-band filter is shown in more detail in Figure 9. There are seven processes in the Polyphase Synthesis Sub-band Filter. These are:

1. Input 32 new Sub-band samples, which loads the next 32 sub-band samples into the Polyphase Synthesis Sub-band filter;
2. Shifting the stored matrix values up 64 positions in the 1024 element array V;
3. Matrixing the new 32 sub-band samples by the $N_{ik}$ filter coefficients to give 64 elements, which are loaded into the lower 64 elements of the 1024 element array V;

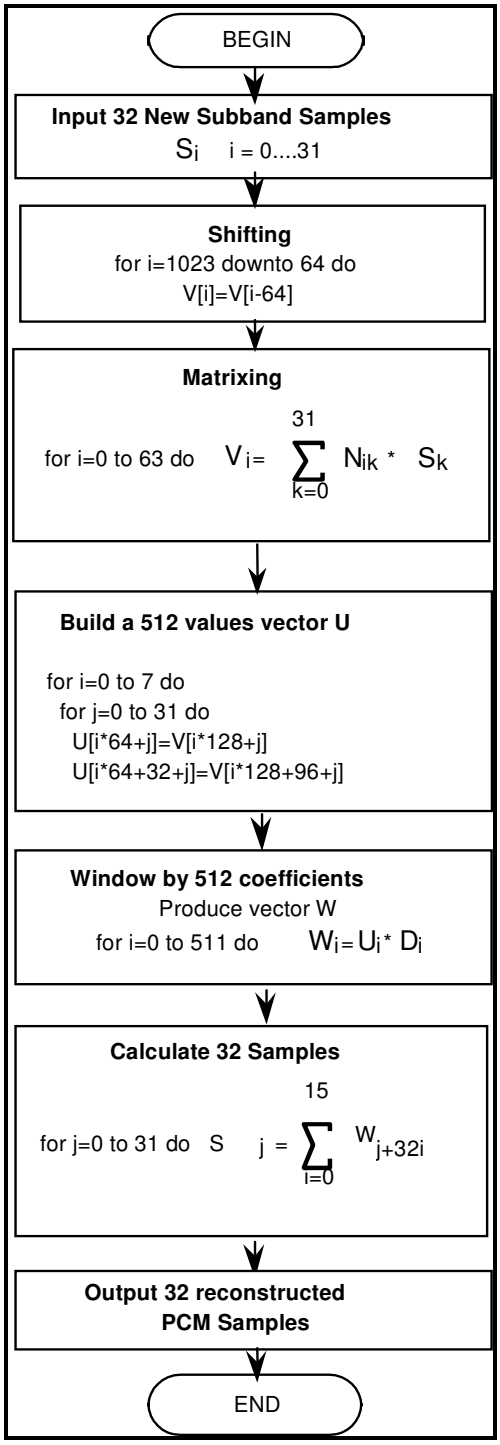


***Figure 9*** *Synthesis sub-band filter flow chart*

4. Building a 512 value vector U which builds an array of 512 values referred to as vector U, from the 1024 element array V;

5. Windowing by 512 coefficients that windows the 512 element array vector U, with the Di filter coefficients to produce the 512 value vector W;
6. Calculate 32 Samples which calculates the 32 decompressed audio samples;
7. Output the 32 reconstructed PCM samples to output the decoded and decompressed audio data.

### 2.11.1 The Shifting of the Polyphase Synthesis Sub-band Filter

In the MPEG standard after the Polyphase Synthesis Sub-band Filter receives the 32 sub-band samples, (6) is applied to an array of 1024 matrixed values called *V* (calculated in the second process described above).

**for $i$ = 1024 down to 64**

$$V[i] = V[i-64] \tag{6}$$

where *V* are the 1024 elements of the array and *i* is the index to the array of *V*.

This has the effect of shifting the lower 960 elements to the upper 960 thereby destroying the upper-most 64 elements and leaving a space of 64 empty elements at the bottom of the array. This "Shifting" stage is shown in the second process of Figure ***9***.

### 2.11.2 The Matrixing of the Polyphase Synthesis Sub-band Filter

The matrixing of the requantised compressed audio sub-band samples by the $N_{ik}$ filter coefficients, gives 64 matrix values which are loaded into the lower 64 elements of the 1024 element array *V*. This is achieved by using the formula

**for i = 0 to 63**

$$V(i) = \sum_{k=0}^{31} (N_{ik} \times S_k) \tag{7}$$

where *V(i)* are the lower 64 elements of the matrixed array, $N_{ik}$ are the matrixing coefficients of the Polyphase Synthesis Sub-band Filter, $S_k$ are the 32 sub-band samples, *i* is the index to the lower 64 elements the array of *V* and *k* is the index to the sub-band samples. The *Nik* matrixing coefficients are calculated from (8) which generates a table of 2048 $N_{ik}$ filter coefficients.

**for i = 0 to 63; k = 0 to 31**

$$N_{ik} = \cos[(16 + i)(2k + 1)(\pi / 64)] \tag{8}$$

where $N_{ik}$ are the matrixing coefficients of the Polyphase Synthesis Sub-band Filter, *i* is the index to the lower 64 elements of the matrixing coefficients of the Polyphase Synthesis Sub-band Filter, *k* is the index to the 32 sub-band samples and $\pi$ is the mathematical constant.

Once the 64 matrix values have been calculated, they are stored in the 64 empty elements at the bottom of the array of matrixed data values called *V*. This "Matrixing" stage is shown in the third process of Figure 9.

### 2.11.3 Building the Vector Table U

When 32 new matrixed values have been stored in the 1024 element array *V*, the Polyphase Synthesis Sub-band Filter uses *V* to build a 512 value array called the vector table *U*. This achieved with (9)

**for i = 0 to 7**
  **for j = 0 to 31**

$$U(i \times 64 + j) = V(i \times 128 + j)$$
$$U(i \times 64 + 32 + j) = V(i \times 128 + 96 + j) \quad (9)$$

where *U* is the 512 value array of vector data, *V* is the 1024 value array of matrixed data, and *i* and *j* are counters for indexing the *U* and *V* arrays. This reduces the 1024 elements of the matrix array *V*, to an array of 512 values in a vector *U*. This is shown in the fourth process of Figure 9.

### 2.11.4 Windowing by 512 Coefficients

After the Polyphase Synthesis Sub-band Filter builds the 512 value vector *U*, it windows it by multiplying by the 512 array of Polyphase Synthesis Sub-band Filter coefficients, referred to as $D_i$, to form a 512 value array referred to as the window vector *W* as shown in (10) (see Figure 9)

**for i = 0 to 511**

$$Wi = Ui \times Di \quad (10)$$

where *W* is the 512 value array of windowed data, *U* is the 512 value array of vectored data, *D* are the filter coefficients that are given in the ISO/IEC 11172 standard and *i* is the index to the windowed data, the vectored data and the filter coefficients.

### 2.11.5 Calculating the Decompressed Audio Sample

The final step of the filter process shown in the sixth process of Figure ***9***, involves the calculation of the 32 samples of decompressed audio data. These are derived by summing every 32nd sample as described by (11):

**for j = 0 to 31**

$$S_j = \sum_{i=0}^{15} W_{j+32i} \tag{11}$$

where *S* represents the 32 decompressed audio samples, *W* is the 512 value array of windowed data, and *i* and *j* are the indexes to the windowed data and the decompressed audio samples.

## 2.12 The Ancillary Data of a MPEG Frame

The fourth part of the MPEG audio frame is reserved for ancillary data such as song title, artist etc. This is user defined. It does not have any effect on the decompression of audio data and will not be further described.

## 2.13 Summary

While there is a wealth of information on variable bit length decoding of compressed video data, there is very little that applies directly to audio decompression, and in particular the tradeoffs between resource utilisation, resolution, accuracy and performance. Research to date is predominantly based on implementation in microprocessors rather than direct hardware synthesis, in VHDL for example. Because of this, existing implementations typically operate with bit lengths fixed at an integer multiple of the microprocessor word size. While it is highly likely that there has been

research into these areas, not a great amount has been published in the open literature. This is almost certainly due to its high commercial value.

The work in this thesis is aimed at investigating the impact of varying the bit resolution of the mathematical operations on the accuracy of the resulting decoded audio. The VHDL model for decoding the MPEG audio implements a variation on the polyphase synthesis sub-band filter, whereby a group of the mathematical operations are performed at the same time, causing a cost penalty in hardware resources, but with the advantage of using far less time.

The ability to decrease the accuracy of the mathematical operations to a certain point, without affecting the final decoded result, can allow a designer to use far less hardware to make a cheaper product. Alternatively, accuracy may be sacrificed to save hardware and thus power. Smaller power usage will lead, in turn, to longer battery life in a portable device.

# Chapter 3. An MPEG 1 Layer I Audio Decoder Design

## 3.1 Introduction

In order to determine the effect of varying the bit lengths on the accuracy of decoded compressed audio, a VHDL model of an MPEG 1 Layer I audio decoder was designed and simulated. The design of the VHDL model of the MPEG 1 Layer I audio decoder was made with reference to the official standard [1]. Where possible, the model was designed in a manner that improved its efficiency—by decreasing the amount of hardware resources required or by reducing the amount of time required for decoding a compressed sample. The VHDL model uses a fractional binary format for the storage of variables and the mathematical operations performed, thus avoiding the need for complex floating point operations.

## 3.2 The MPEG 1 Layer I Audio Decoder Model in VHDL

The VHDL model of the MPEG 1 Layer I audio decoder shown in Figure 10 comprises three major processing blocks along with three look-up tables and some other smaller processing blocks. The major sections are the Header process, the Layer I decoder and the Polyphase Synthesis Sub-band Filter. The look-up tables are called the `bit_lut`[2], the `N_lut` and the `valid_lut`. The remaining parts are the clock generator, the reader ROM and the parallel to serial converter. In general terms, the Header process decodes the header information contained in the audio frame. The Layer I decoder is

[2] Note: in this and the following chapters, a convention has been adopted that uses `Courier` font to designate file names or VHDL components such as signal or variables.

used to decompress the compressed audio data into requantised sub-band samples, which are then passed to the Polyphase Synthesis Sub-band Filter which transforms them into audio samples.

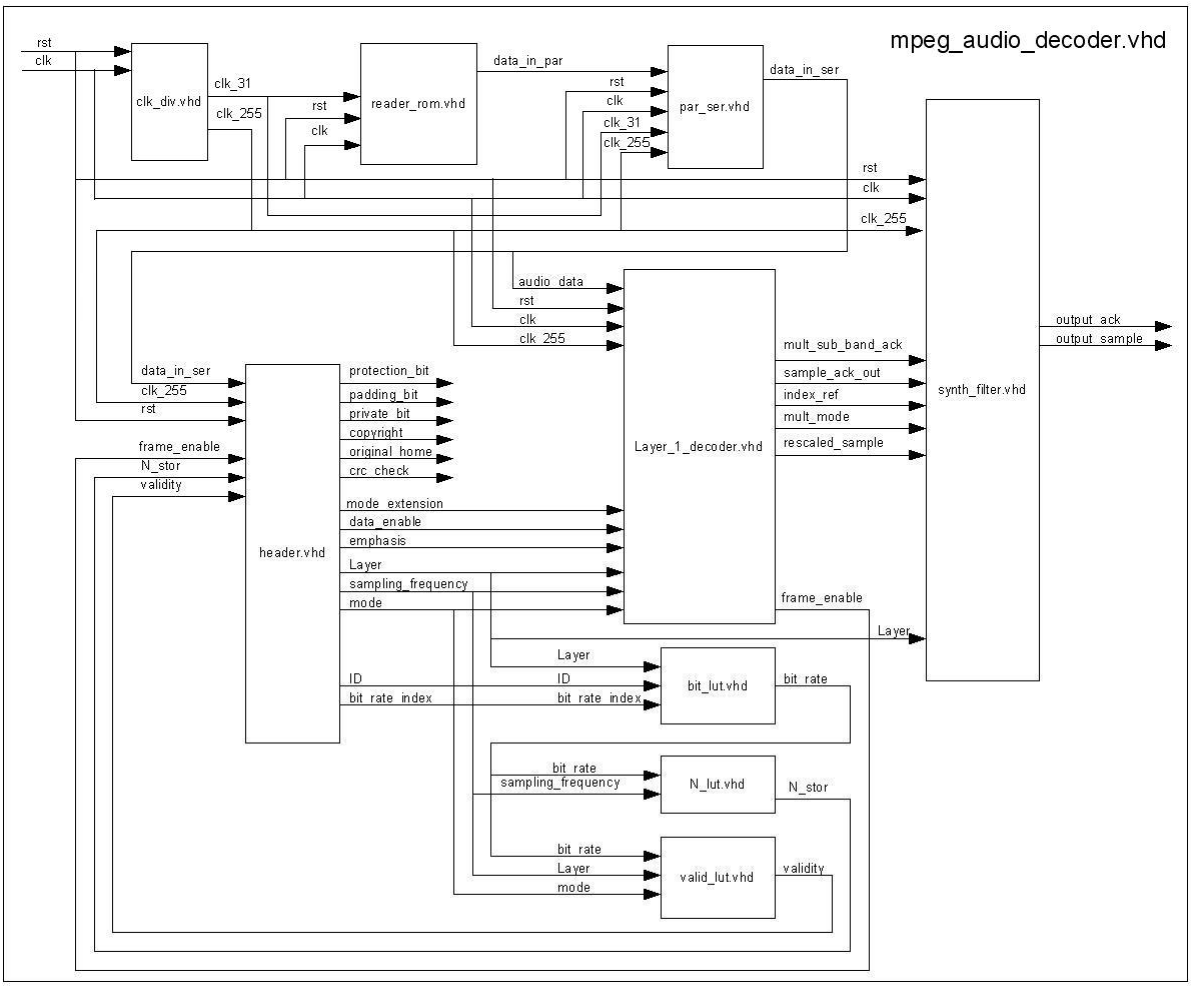


***Figure 10*** *Block diagram of VHDL MPEG model*

## 3.3 The Clock Generator

The Clock Generator is shown in the file of VHDL code shown in **Appendix 1-1** and called `clk_div.vhd`. Its function, shown as a flowchart in Figure 11 is to act as a clock divider thereby generating the clock signals `clk_255` and `clk_31` required for

the audio decoder model. The initial clock signal is supplied from test bench and is referred to as `clk`.

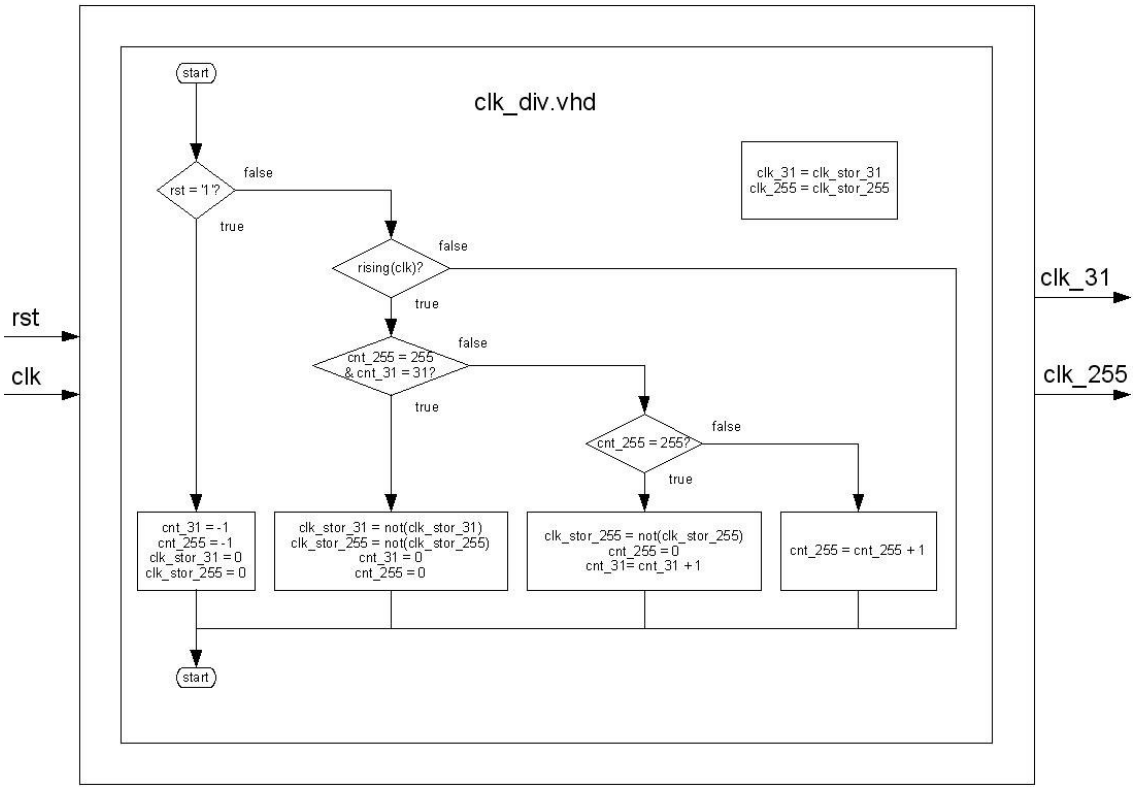


***Figure 11*** *Flowchart of clk_div.vhd*

The clock signal `clk_255` divides the basic clock signal (`clk`) by 512 and `clk_31` by a further factor of 64, together generating a clock division of 32,768.

## 3.4 The Reader ROM

On each rising edge of the clock signal `clk_31` the Reader ROM (**Appendix 1-2**, `reader_rom.vhd`) reads in a 32 bit wide data words of compressed audio data from the file (`fl4.mp3`), originally sourced from the standard [2]. In a commercial product, this compressed audio would be sourced from either a type of digital storage medium such as a hard drive or flash memory, or a communications receiver device such as a

radio tuner or modem and this process would replace this section of VHDL code. The data is then passed on to the Parallel to Serial Converter.

## 3.5 Parallel to Serial Converter

The Parallel to Serial Converter `par_2_ser.vhd` in **Appendix 1-3** (see also the flowchart in Figure 12) accepts a 32 bit wide data word from the Reader Rom representing part of a compressed MPEG 1 Layer I audio data frame. On each rising edge of the clock `clk_255`, the Parallel to Serial Converter passes one bit of the 32 bit wide data word to the Header and the Layer 1 decoder.

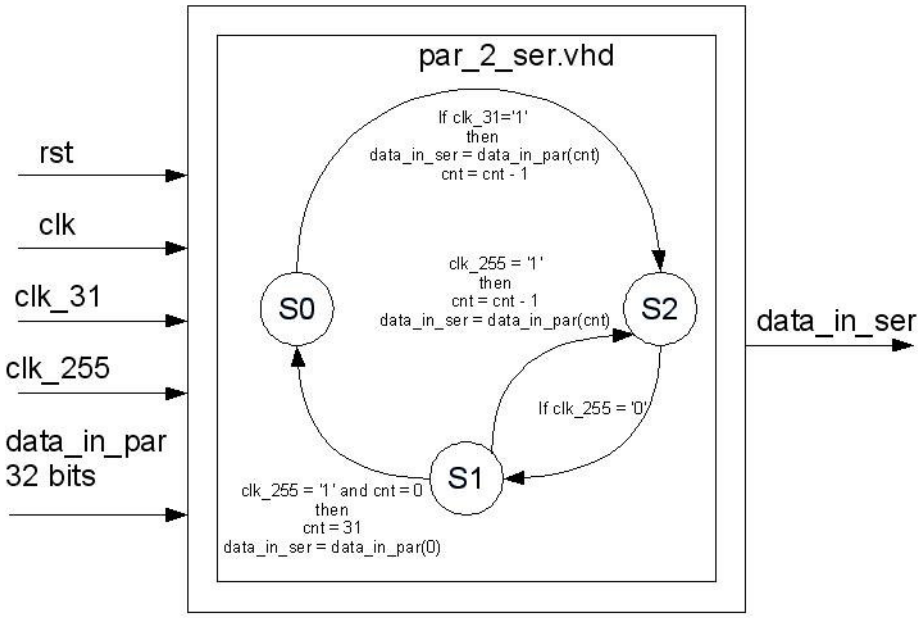


**Figure 12** *Flowchart of par_2_ser.vhd*

## 3.6 The Header

As just mentioned, the header receives a single bit of the audio data frame from the Parallel to Serial Converter, at each rising edge of its clock. There are, at most, four

different types of data present in MPEG 1 Layer I audio frames: the header; the optional error check; the compressed audio data and the optional ancillary data.

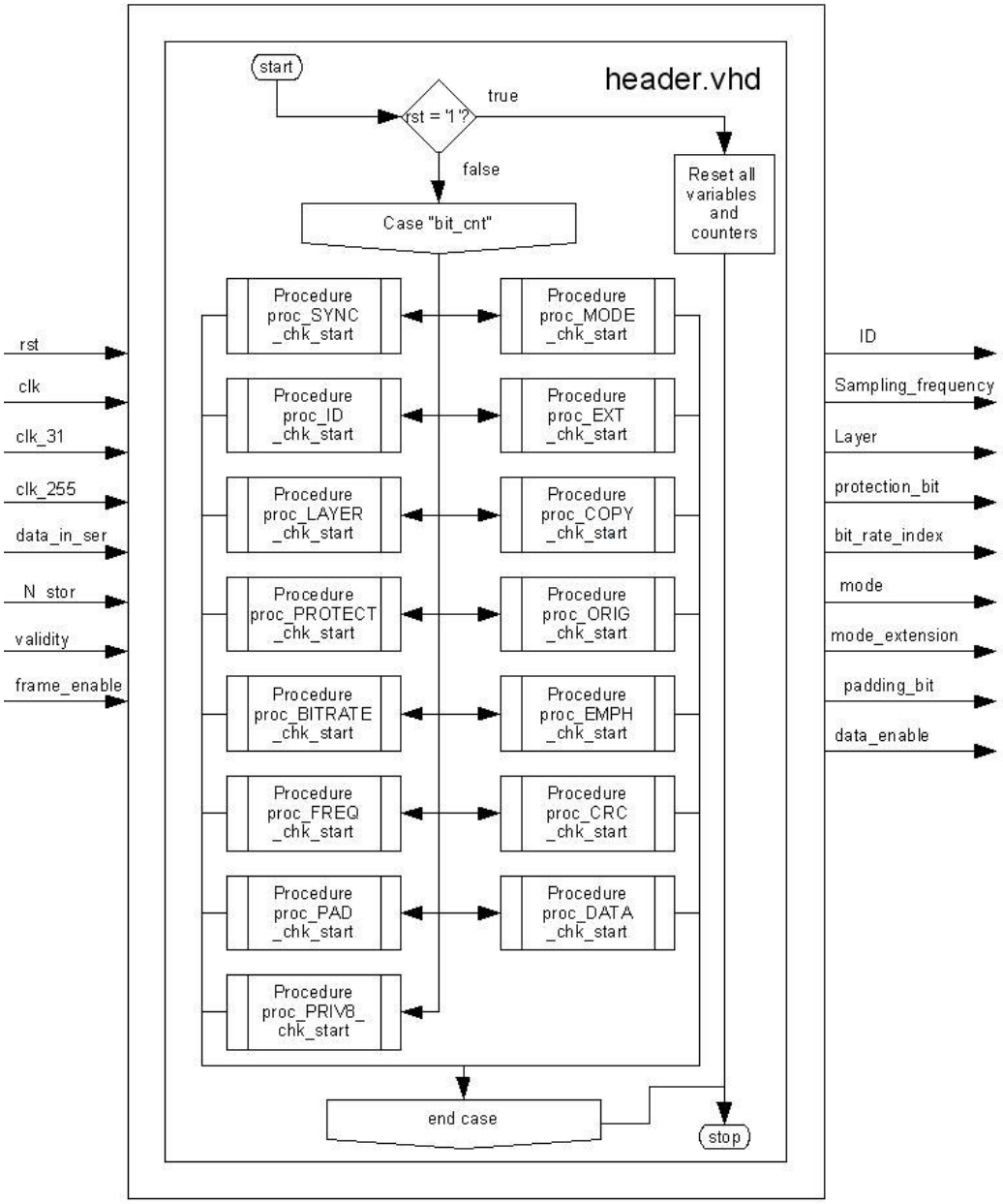


***Figure 13*** *Flowchart of header.vhd*

The header block is implemented as a state machine using a combination of a case statement and procedures (**Appendix 1-4**, `header.vhd`). Each of the different elements of data has its own state modelled with a procedure. These procedures are shown in Figure 13 and operate as follows:

**proc_SYNC_chk_Start**

This procedure (Figure 14) finds the sync word by counting 11 consecutive high bits of data (i.e., 111,1111,1111) from the MPEG audio frame via the Parallel to Serial Converter. Each time a high bit (i.e., 1) is received, the counter signal `cnt` is incremented by one. The signal `bit_cnt` is incremented to the value 12 causing the case statement to enter the state of procedure `proc_ID_chk_Start`.

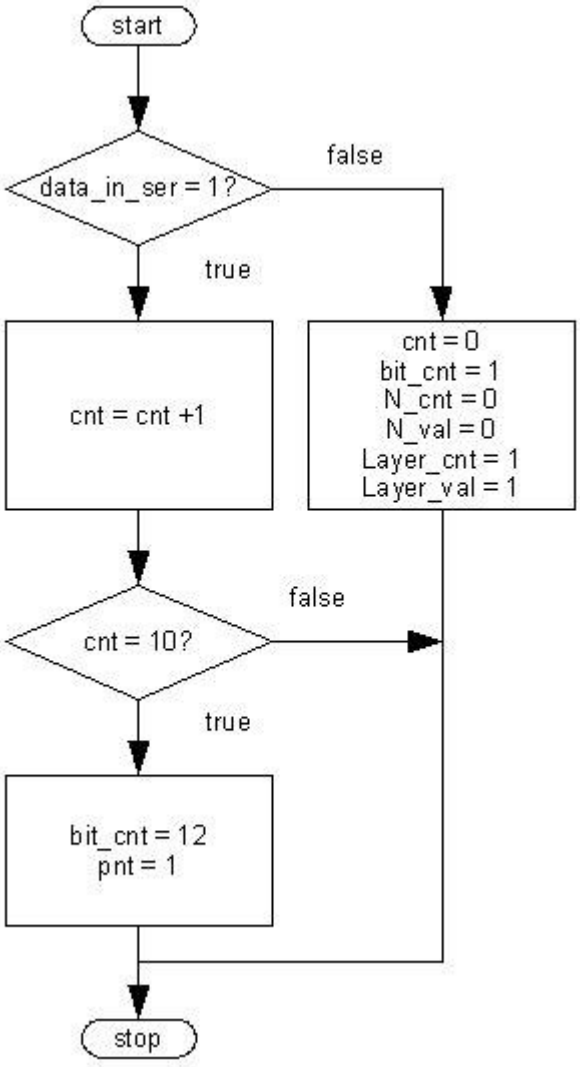


***Figure 14*** *Flowchart of proc_SYNC_chk_Start*

**proc_ID_chk_Start**

The two identifier bits read by this procedure (Figure 15) signify whether the following data is from a MPEG audio frame and, if so, its type (e.g., MPEG 1, MPEG 2 or MPEG

2.5 – see Table 1). The value of the pointer signal `pnt` is decremented as each bit of the ID is read. The signal vector `ID_stor` stores the ID value. If either bit of the signal vector `ID_stor` is 0, the identifier bits are set to a version of MPEG other than 1, or the MPEG audio frame is corrupt and the `bit_cnt` is set to 1, forcing the case statement to call the procedure `proc_SYNC_chk_Start` and start looking for the next sync word. When the second bit is read the pointer signal `pnt` is set to 1 (to count the two bits of the Layer switch in the next procedure) and the `bit_cnt` is incremented to the value 14 causing the case statement to enter the state of procedure `proc_Layer_chk_Start`.

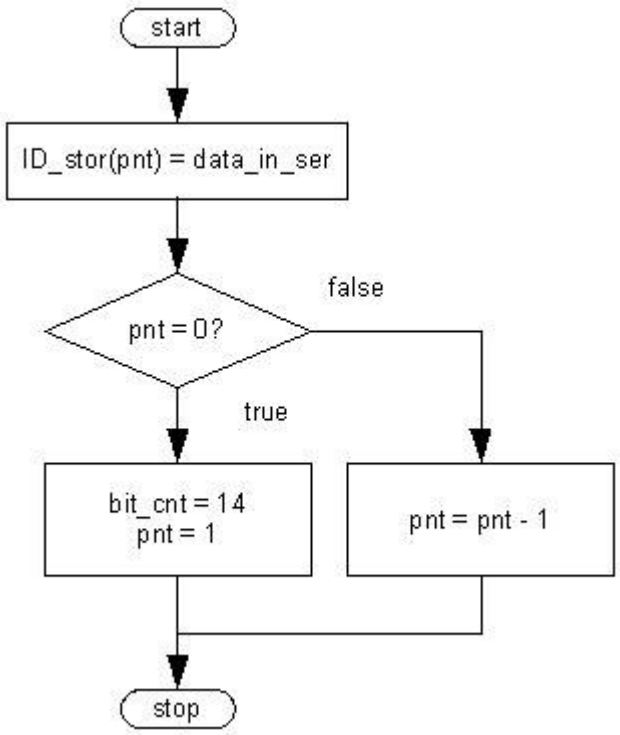


***Figure 15*** *Flowchart of proc_ID_chk_Start*

**proc_LAYER_chk_Start**

This procedure (Figure 16) obtains the value of the two bits for the layer switch, which describes the layer of compression that has been used for the compressed audio data in the MPEG 1 audio frame (e.g., Layer I, II or III as shown in Table 2). At the start of

procedure an error check is performed on the two bit signal vector `ID_stor` to see if it equals 01 (the reserved value). If this occurs, the MPEG audio frame is considered to be corrupt and the `bit_cnt` is set to 1, forcing the case statement to call the procedure `proc_SYNC_chk_Start` to start looking for the next sync word. If a valid layer switch is read the pointer signal `pnt` is set to 3 (to count the four bits, from 3 down to 0, of the bit rate switch in a following procedure) and the `bit_cnt` is incremented to the value 16 causing the case statement to `proc_PROTECT_chk_Start`.

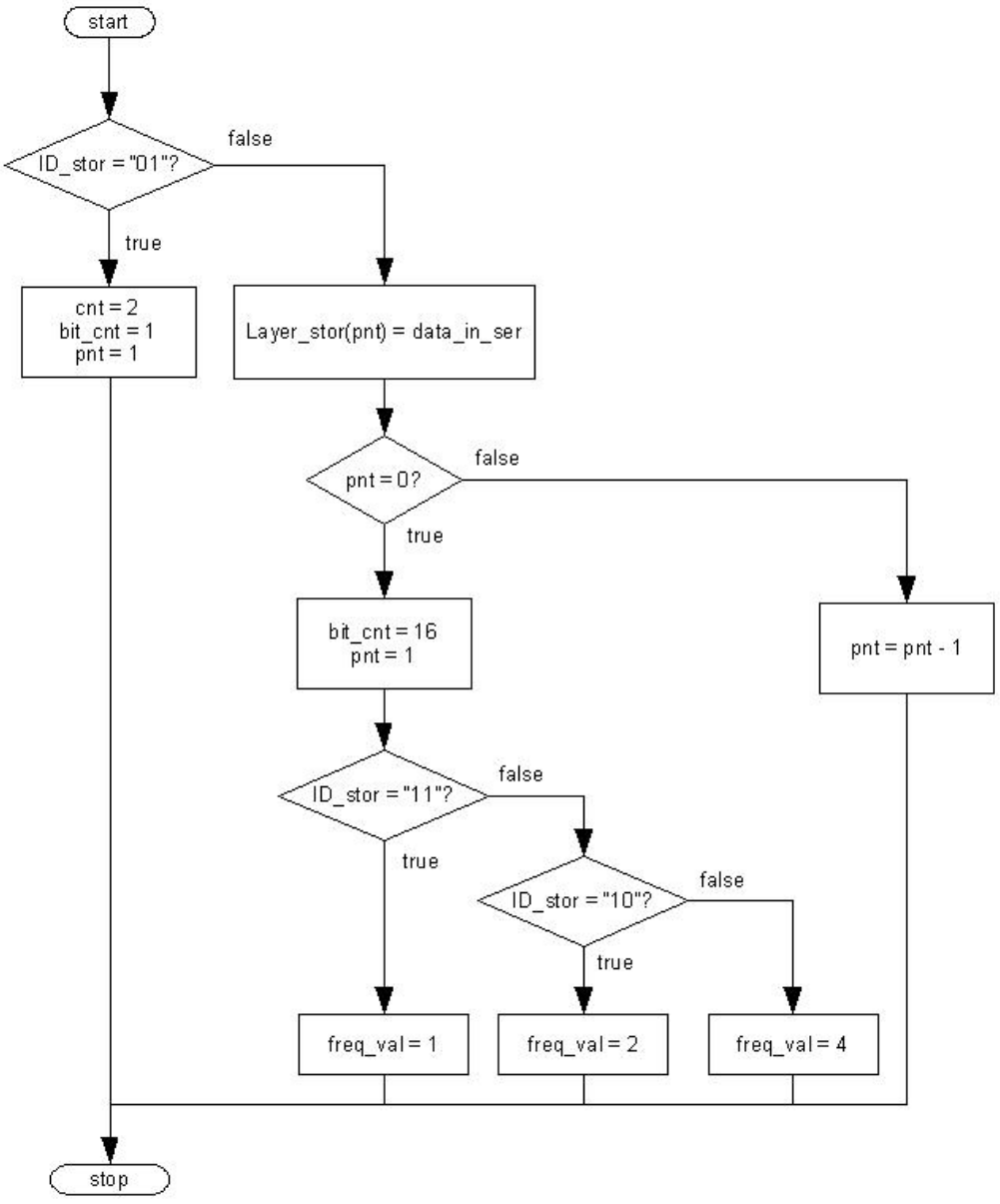


***Figure 16*** *Flowchart of proc_LAYER_chk_Start*

**proc_PROTECT_chk_Start**

The purpose of this procedure (Figure 17) is to read the protection bit, which is used to determine whether there is data correction present in the MPEG 1 audio frame. If the protection bit is 0 error correction has been added to the MPEG 1 audio frame, and a further 16 bits will have to be counted after the header part of the MPEG 1 frame. If the protection bit is 1, error correction has not been added and the header will be followed by compressed audio data. At the start of this procedure `Layer_stor` is checked to see if it is 00 as this is reserved as shown in Table 2. If this happens the count variable signal `bit_cnt` is set to 1, forcing the case statement to call the procedure `proc_SYNC_chk_Start` and start looking for the next sync word. When the protection bit is read the pointer signal `pnt` is set to 3 (to count the four bits of the bit rate switch in the following procedure) and the `bit_cnt` is incremented to the value 17 causing the case statement to exit this procedure, and enter the state of procedure `proc_BITRATE_chk_Start`.

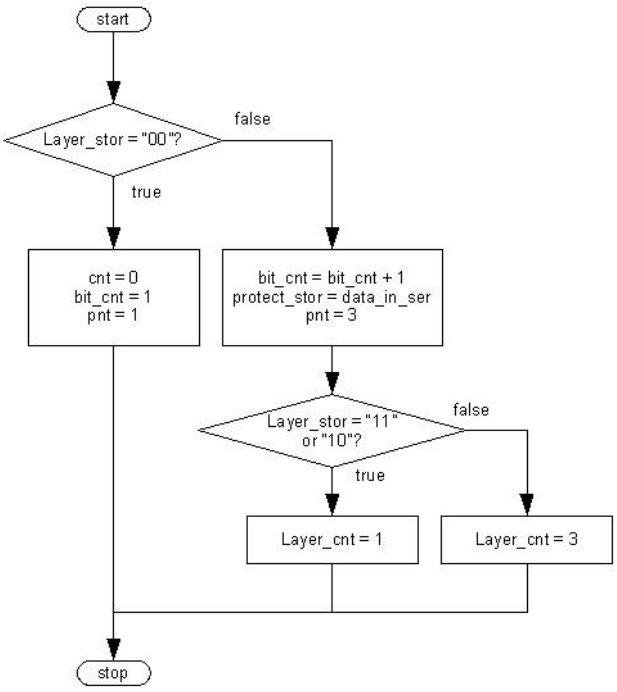


***Figure 17*** *Flowchart of proc_PROTECT_chk_Start*

**proc_BITRATE_chk_Start**

This procedure (Figure 18) reads in the four bits of the bit rate index that are applied to the bit rate look up table to determine the bit rate (see Table 3), and which also determines the number of slots (NOS) of data in a frame of MPEG audio data (described below). When the fourth bit is read the pointer signal `pnt` is set to 1 (to count the two bits of the sampling frequency switch in the following procedure) and the `bit_cnt` is incremented to the value 21 which causes the case statement to transfer to the procedure `proc_FREQ_chk_Start`.

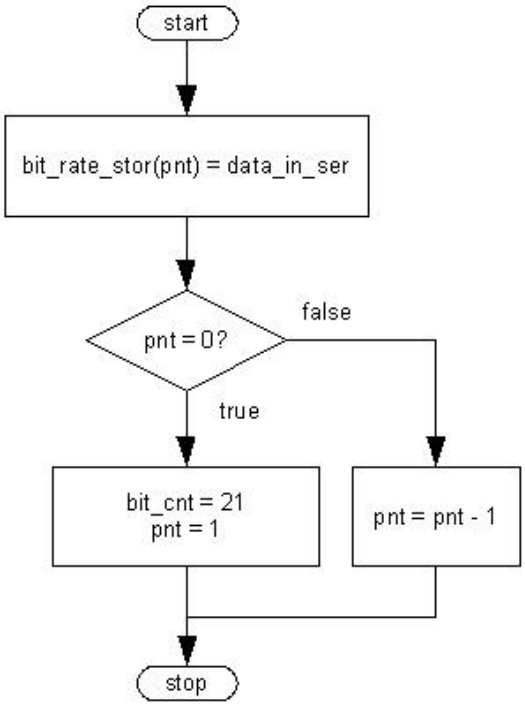


***Figure 18*** *Flowchart of proc_BITRATE_Start*

**proc_FREQ_chk_Start**

The two bits of the sampling frequency switch read by this procedure (Figure 19) determine the sampling frequency of the compressed audio following (see Table 4). At the start of procedure the four bit signal vector `bit_rate_stor` is checked to see if it equals 1111 as this is reserved (see Table 3). In that case, the data from the MPEG audio frame is either corrupt or not an MPEG audio frame and thus the count variable

signal bit_cnt is set to 1, forcing the case statement to call the procedure procedure proc_SYNC_chk_Start and start looking for the next sync word. When the two bits are assigned to the two bit signal vector freq_stor, the pointer signal pnt is set to 1 (to count the two bits of the mode switch in a following procedure) and the bit_cnt is incremented to the value 23 which causes the case statement to exit this procedure and enter proc_PAD_chk_Start.

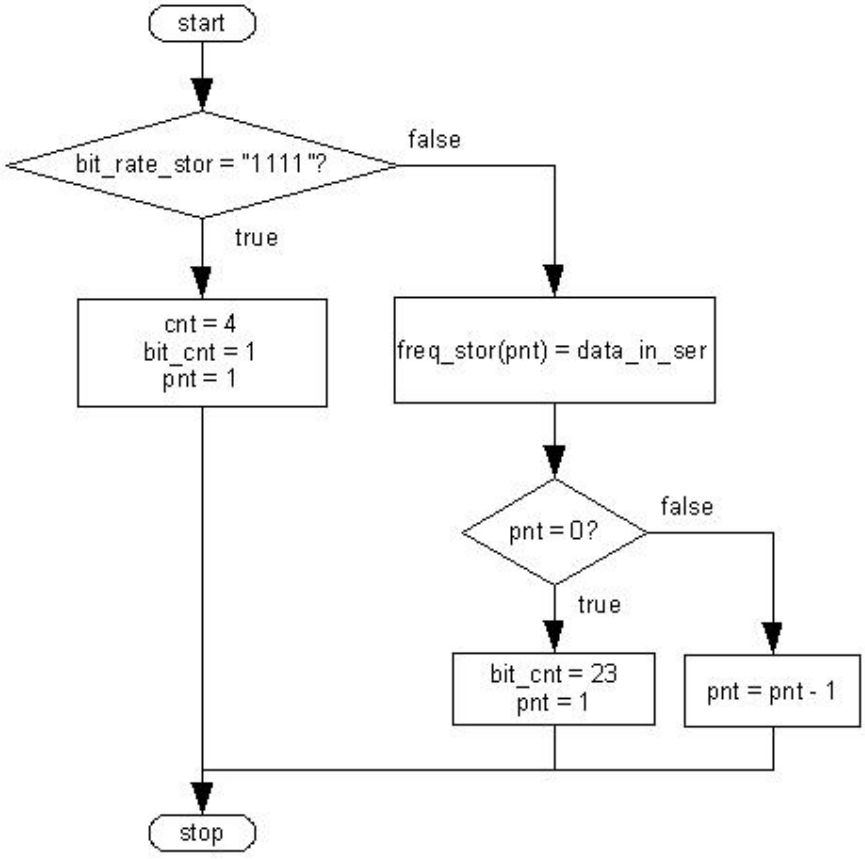


***Figure 19*** *Flowchart of proc_FREQ_chk_Start*

**proc_PAD_chk_Start**

The procedure proc_PAD_chk_Start (Figure 20) is used to obtain the padding bit, which if set high adds an extra slot of data before the next sync word. At the start of procedure the two bit signal vector freq_stor is checked to see if it equals the reserved value of 11 (see Table 4) meaning that the data from the MPEG audio frame is

corrupt, or not an MPEG audio frame. If this happens the count variable signal `bit_cnt` is set to 1, forcing the case statement to call the procedure `proc_SYNC_chk_Start` and start looking for the next sync word. When the pad bit is read the pointer signal `pnt` is set to 1 and the `bit_cnt` is incremented to the value 24. The pointer signal `pnt` is set to 1 (to count the two bits of the mode switch in a following procedure) and the `bit_cnt` incremented to the value 24 causing the case statement to transfer to `proc_PRIV8_chk_Start`.

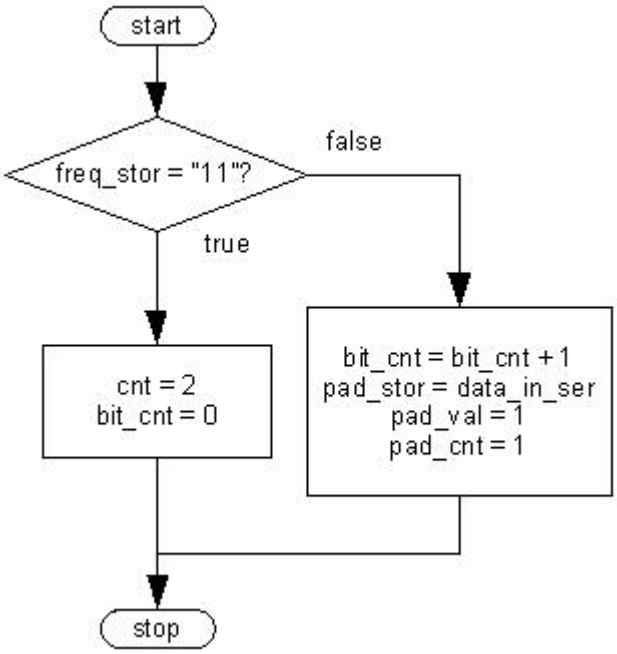


***Figure 20*** *Flowchart of proc_PAD_chk_Start*

**proc_PRIV8_chk_Start**

The private bit is no longer used but has to be included to remain backward compatible to earlier versions of the MPEG header frame format. At the start of the procedure (Figure 21) the padding bit is checked. If it is set, this means that padding is required and the two-bit Layer switch determines its value. Then the the two bit sampling frequency switch is checked to see if it equals 00, representing 44.1kHz (see Table 4), and the four bits of the bit rate is checked to see if it equals 0000, representing free

format (Table 3). As this is invalid in this case, the procedure exits and starts looking for the next sync word. When the bit is read the pointer signal `pnt` is set to 3 (in preparation for reading the four bits of bit rate in the following procedure) and the `bit_cnt` is incremented to the value 25 which causes the case statement to transfer to `proc_MODE_chk_Start`.

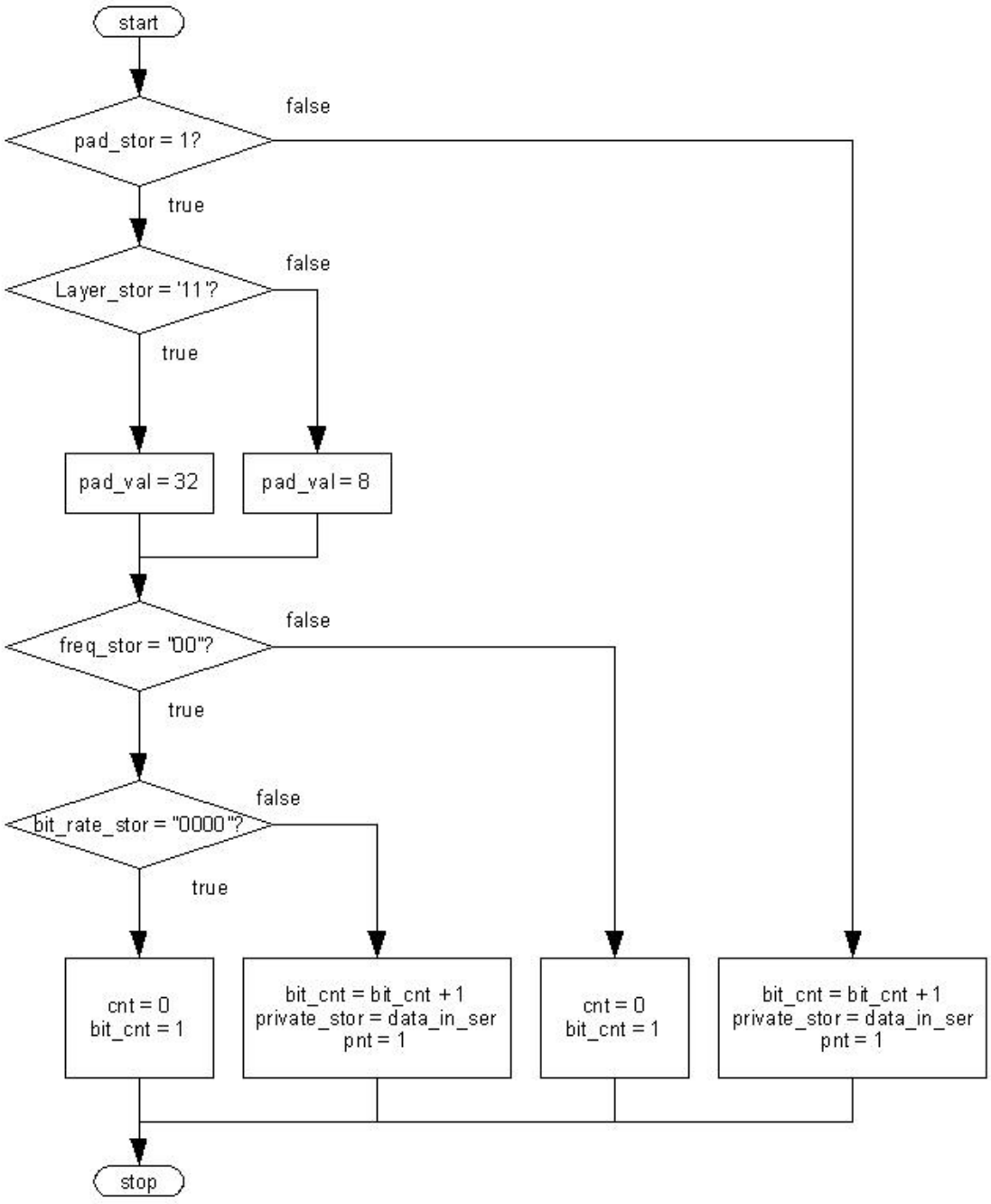


***Figure 21*** *Flowchart of proc_PRIV8_chk_Start*

**proc_MODE_chk_Start**

This procedure (Figure 22) reads the two bits of the mode switch (see Table 5). When the second bit is read the pointer signal `pnt` is set to 1 (for reading the two bits of extension mode in the following procedure), and the `bit_cnt` is incremented to the value 27 causing the case statement to transfer to `proc_EXT_chk_Start`.

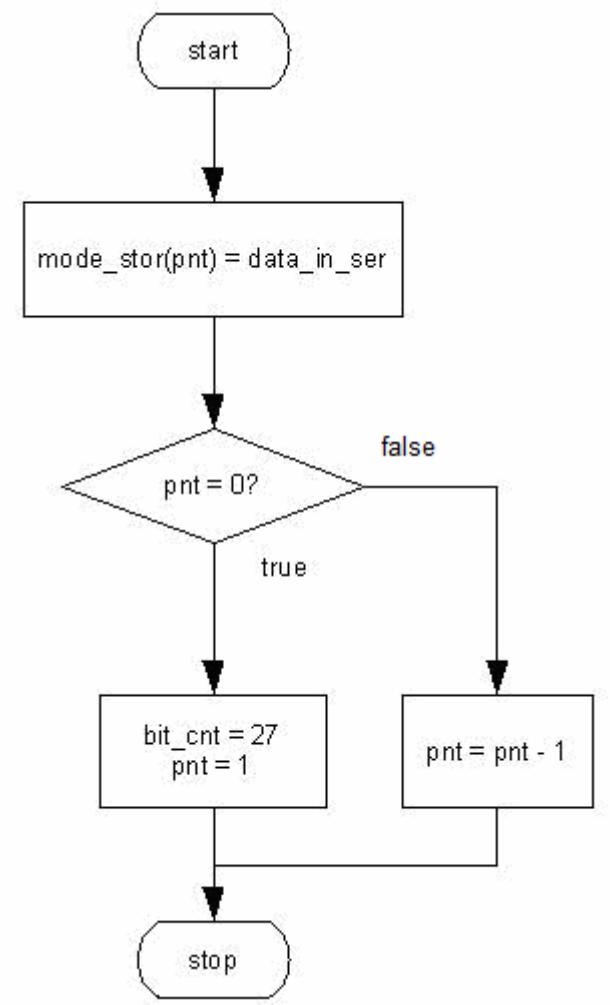


***Figure 22*** *Flowchart of proc_MODE_chk_Start*

**proc_EXT_chk_Start**

The two bits of the mode extension switch read in this procedure (Figure 23) set the sub-band limits in joint stereo mode (see Table 6). The signal `validity` is checked to see if it is set. The `validity` signal represents whether the combination of the MPEG version, the Layer, the mode and the bit rate of the MPEG audio frame form a

valid combination and is supplied by the validity look up table. If not valid the counter signal `bit_cnt` is set to 1 and the decoder starts looking for the next sync word. If it is valid, then the mode extension bits will be read from the MPEG audio frame, the pointer signal `pnt` is set to 1 (for reading the two bit signal emphasis in a following procedure) and `bit_cnt` set to the value 29 causing the case statement to to `proc_ORIG_chk_Start`.

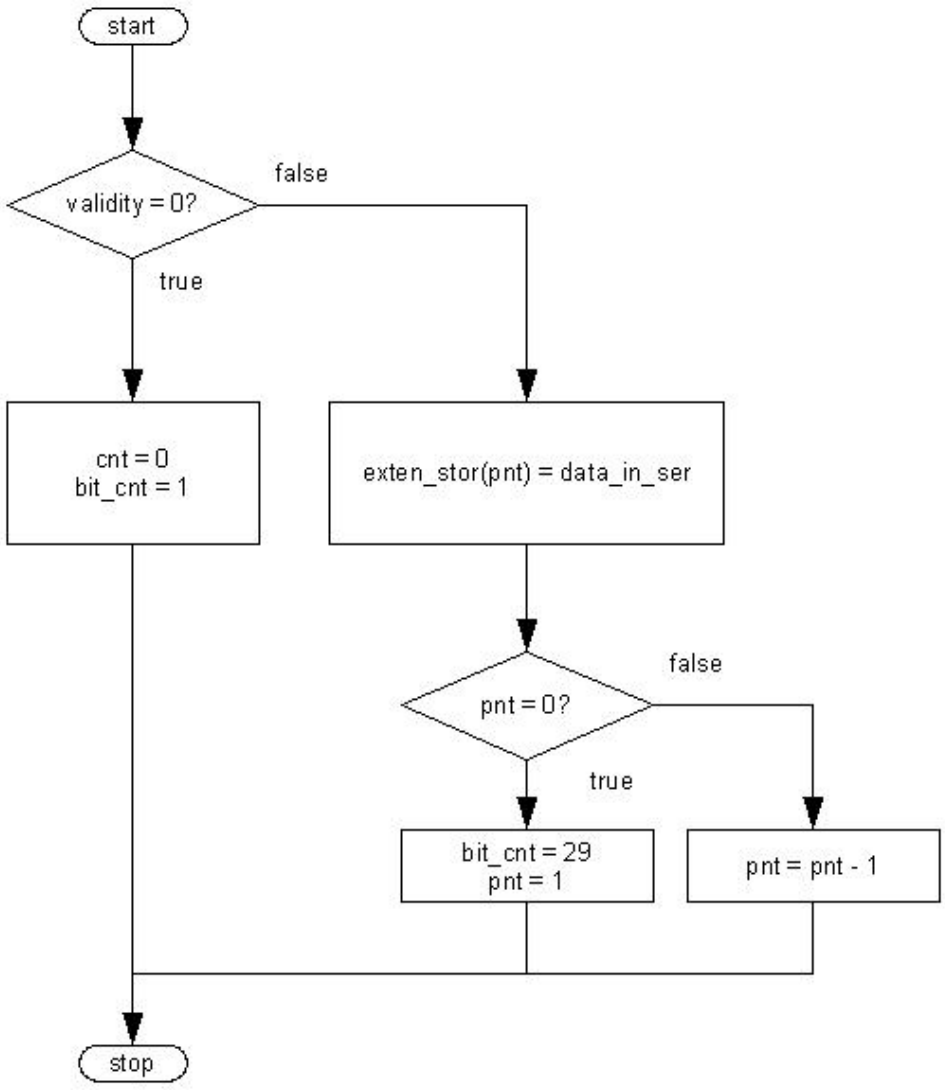


***Figure 23*** *Flowchart of proc_EXT_chk_Start*

**proc_COPY_chk_Start**

This procedure (Figure 24) reads the copyright bit which determines if the compressed audio data in the MPEG 1 audio frame is copyright protected. When the bit is read the pointer signal `pnt` is set to 1 (in preparation for reading the two bit signal emphasis in a following procedure) and the `bit_cnt` is incremented to the value 30 which causes the case statement to exit to `proc_ORIG_chk_Start`.

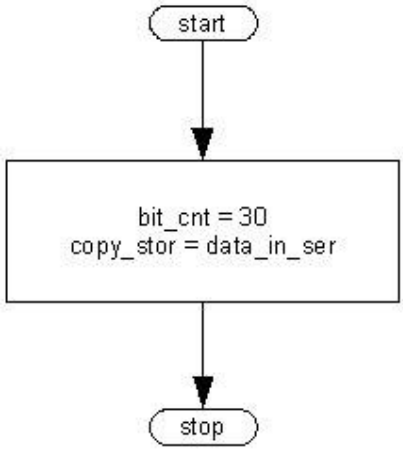


***Figure 24*** *Flowchart of proc_COPY_chk_Start*

**proc_ORIG_chk_Start**

This procedure (Figure 25) reads the original bit to determines if the compressed audio data in the MPEG 1 audio frame is original or a copy. When the bit is read, the pointer signal `pnt` is set to 1 (in preparation for reading the two bit signal emphasis in the following procedure), and the `bit_cnt` is incremented to the value 31 causing the case statement to exit to `proc_EMPH_chk_Start`.

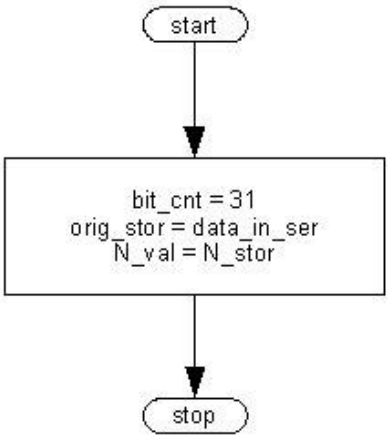


***Figure 25*** *Flowchart of proc_ORIG_chk_Start*

**proc_EMPH_chk_Start**

The two bits read by this procedure (Figure 26) read the emphasis switch determining the level of de-emphasis of the compressed audio data (see Table 7). Following the second bit, the value of the stored protection bit is checked to see if the protection bit is set. If it is not, the MPEG audio frame contains a cyclic redundancy check and the pointer signal `pnt` is set to 15 (for reading the 16 bit signal for cyclic redundancy check) and the `bit_cnt` is incremented to the value 33 causesing the case statement to exit to `proc_CRC_chk_Start`. If the protection bit stored in the signal `protect_stor` is set, the MPEG audio frame does not contain a cyclic redundancy check and the pointer signal `pnt` is set to 1 (for reading the two bit signal vector `ID_stor` in the procedure `proc_ID_chk_Start` at the start of the next MPEG audio frame) and the `bit_cnt` is incremented to the value 49, causing the case statement to exit to `proc_DATA_chk_Start`.

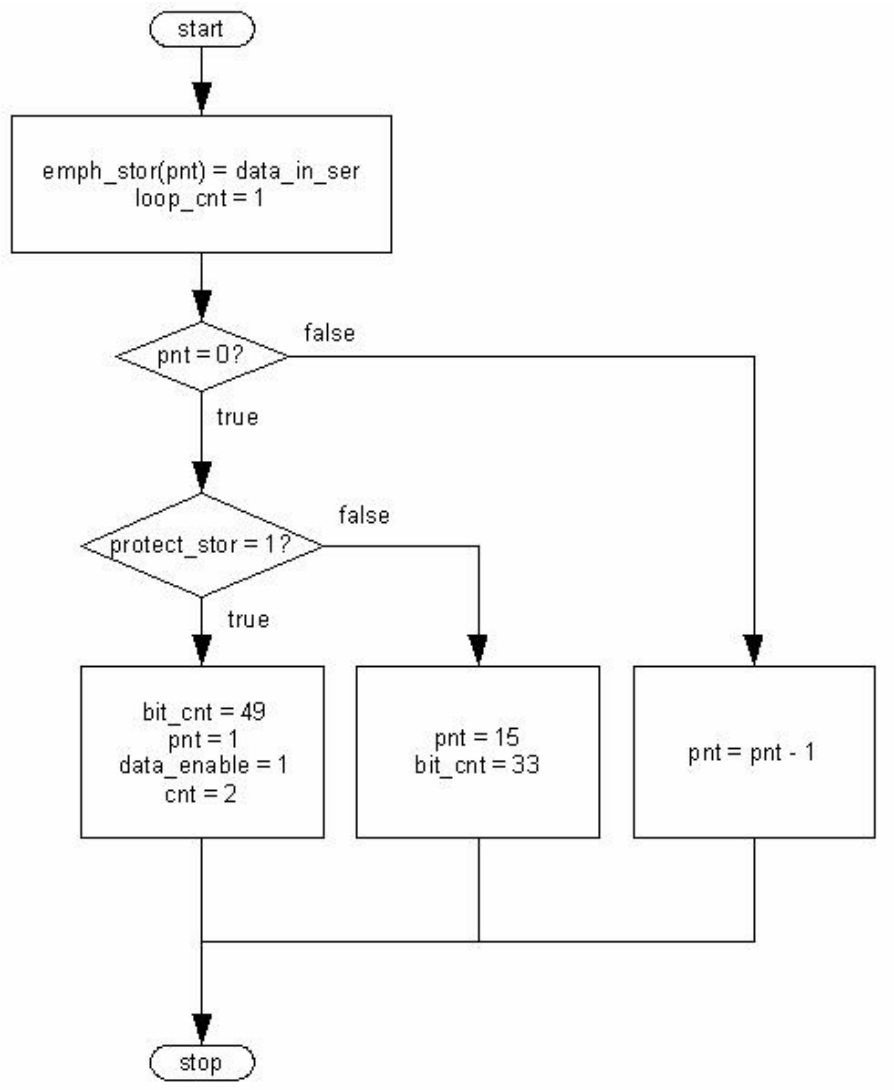


***Figure 26*** *Flowchart of proc_EMPH_chk_Start*

**proc_CRC_chk_Start**

This procedure (Figure 27) examines the 16 bits of the cyclic redundancy check. This is optional and these bits are only read if the protection bit in the header of the MPEG 1 audio frame is cleared. These 16 bits represent cyclic redundancy code of the MPEG audio frame header and are used as a form of error correction. Even though the VHDL model will not use the error correction provided by the cyclic redundancy code, it has to count the number of bits used for the cyclic redundancy code so as to be able to find

the next sync word for the next MPEG frame of data. At the start of the procedure the two bit signal vector `emph_stor` is checked to see if it equals 10. As this is a reserved code (see Table 7), it implies that the MPEG audio frame is corrupt, or not an MPEG audio frame. If this case, the count variable signal `bit_cnt` is set to 1 and the process exits. On the other hand, if the emphasis is valid then the signal vector `data_enable` is set to 1 to inform the other parts of the VHDL model that the data has been processed by the header and that the Layer I decoder and synthesis filter are to begin processing data. When the 16th bit is read the pointer signal `pnt` is set to 1 (for reading the two bit signal vector `ID_stor` at the start of the next MPEG audio frame) and the `bit_cnt` is incremented to the value 49 causing the case statement to exit.

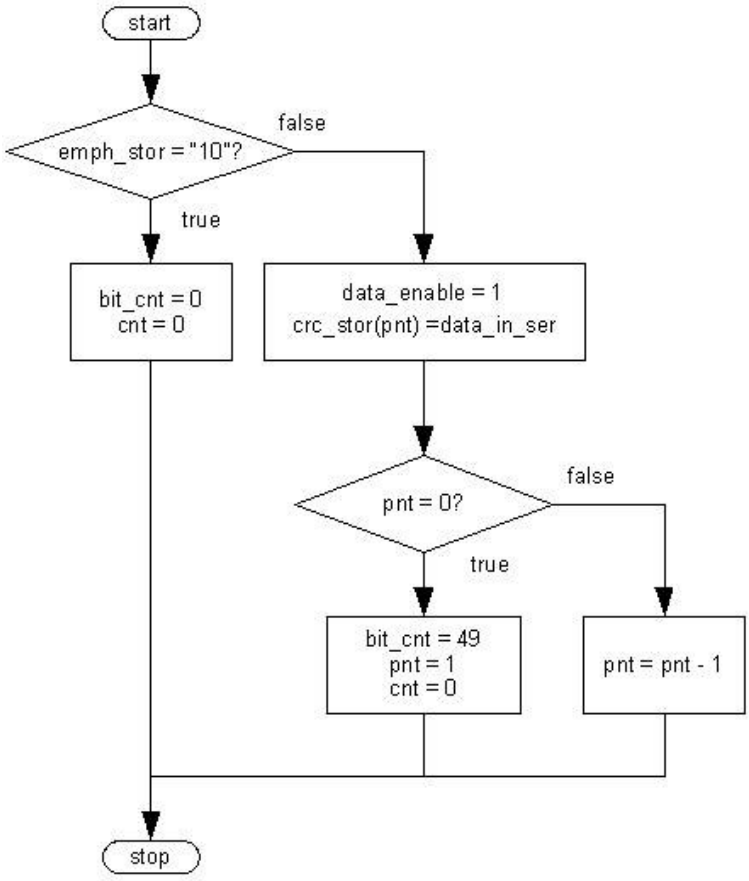


***Figure 27*** *Flowchart of proc_CRC_chk_Start*

**proc_DATA_chk_Start**

This procedure (Figure 28) counts the number of bits present in the compressed audio data in the MPEG 1 audio frame. The number of slots (and therefore the number of bits) in each MPEG 1 audio frame is determined by the bit rate index and the sampling frequency and is supplied by the Number of Slots look up table. This information is used to calculate the number of bits before the next sync word (as shown in **Section 2.9.3**). The procedure checks to see if the signal frame enable is high meaning that the Layer I decoder is ready for data. The procedure will then check if the number of slots is 0. If it isn't the procedure will count down until it is. When it is the count variable signal `bit_cnt` is set to 1, forcing the case statement to call the procedure `proc_SYNC_chk_Start` and start looking for the next sync word.

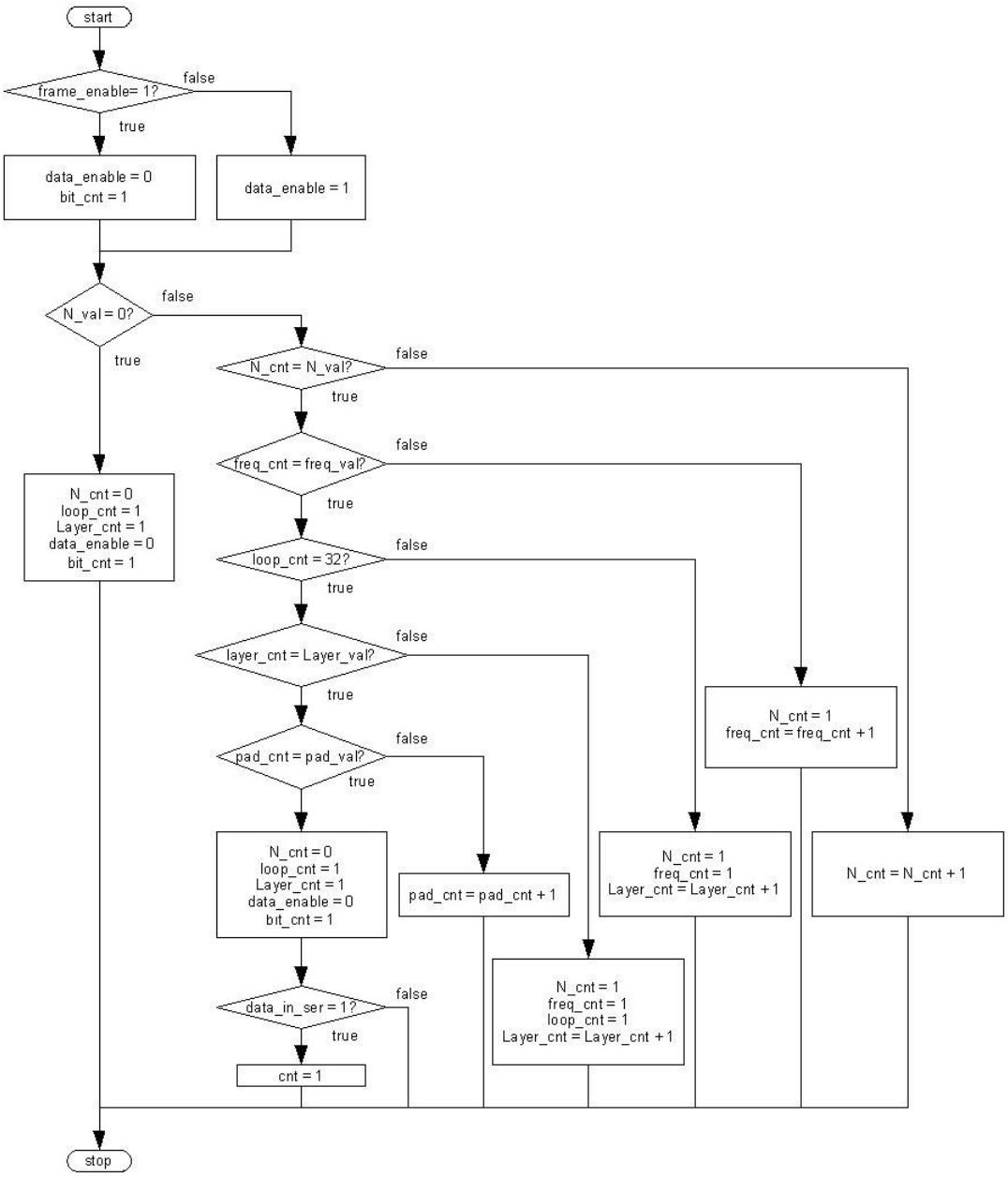


**Figure 28** Flowchart of proc_DATA_Start

## 3.7 The Bit rate look up table

The Bit rate look up table (`bit_lut.vhd`, **Appendix 1-5**) is used to return the bit rate of a frame of MPEG audio data for a given bit rate index. A MPEG audio frame's bit rate is used to allow the VHDL model to determine the size of each MPEG audio

frame, and therefore to find the end of the compressed data section. The table is passed three pieces of information from the header:

1. the identifier bits logic vector `ID`;
2. the Layer switch bits logic vector `Layer`;
3. the bit rate index logic vector `bit_rate_index`.

These are used in a process that returns an integer value called `bit_rate` for every legal combination in the audio frame.

## 3.8 The Validity look up table

The Validity look up table is used to return a valid, or invalid, signal for a MPEG audio frame of data to signify that the frame has a valid combination of Layer, bit rate and mode (see `valid_lut.vhd`, **Appendix 1-6**). The table is passed three pieces of information:

1. the mode switch from the header;
2. the Layer switch vector from the header;
3. the integer `bit_rate` from the Bit rate look up table.

The look up table process returns a logical value of either 0 or 1 for every valid combination. As mentioned above, this logical value is called the `validity` and reflects the validity of the data attributes defined in the header part of a MPEG audio frame.

## 3.9 The Number of Slots look up table

As discussed in **Section 2.9.3**, the compressed audio data in a MPEG frame is divided up into slots. The Number of Slots (NOS) look up table is used to return the number of slots of data in the current MPEG audio frame of data, which is dependent on the sampling frequency and the bite rate. The VHDL model uses the number of slots to determine the amount of data present in each MPEG audio frame, thereby determining the end of the compressed data section of the frame. The Number of Slots look up table is shown in the file of VHDL code shown in **Appendix 1-7** and called `N_lut.vhd`.

The NOS look up table is passed the sampling frequency from the header, plus the bit rate integer in a process that returns the integer value `N_stor` for every valid combination allowed in a MPEG 1 Layer I audio frame.

## 3.10 The Layer I decoder

The compressed audio part of a frame of MPEG audio data is used by the Layer I decoder to decode the samples of compressed sub-band audio samples into decompressed audio sub-band samples. The Layer I decoder shown in **Appendix 1-8** (`Layer_1_dec.vhd`) is organised into a main unit, a multiply unit and three look up tables as can be seen in Figure 29.

There are three types of data present in the compressed audio data part of a MPEG 1 Layer I as follows:

1. the bit allocation of the compressed sub-band sample;

2. the scale factor index;

3. and the compressed sub-band sample.

The main unit of the Layer I decoder assigns to signal vectors the number of bits (referred to as `nb`), the scale factors and the compressed sub-band audio samples stored in the compressed audio section of the MPEG audio frame. The `nb` is used to determine how many bits are read for each compressed sub-band sample. The multiplier unit scales the compressed sub-band sample by the signal vector `nb_mult` (see in Table 12) and by the signal vector `scalefactor` in order to calculate a rescaled sub-band sample represented by *s'* shown in (5).

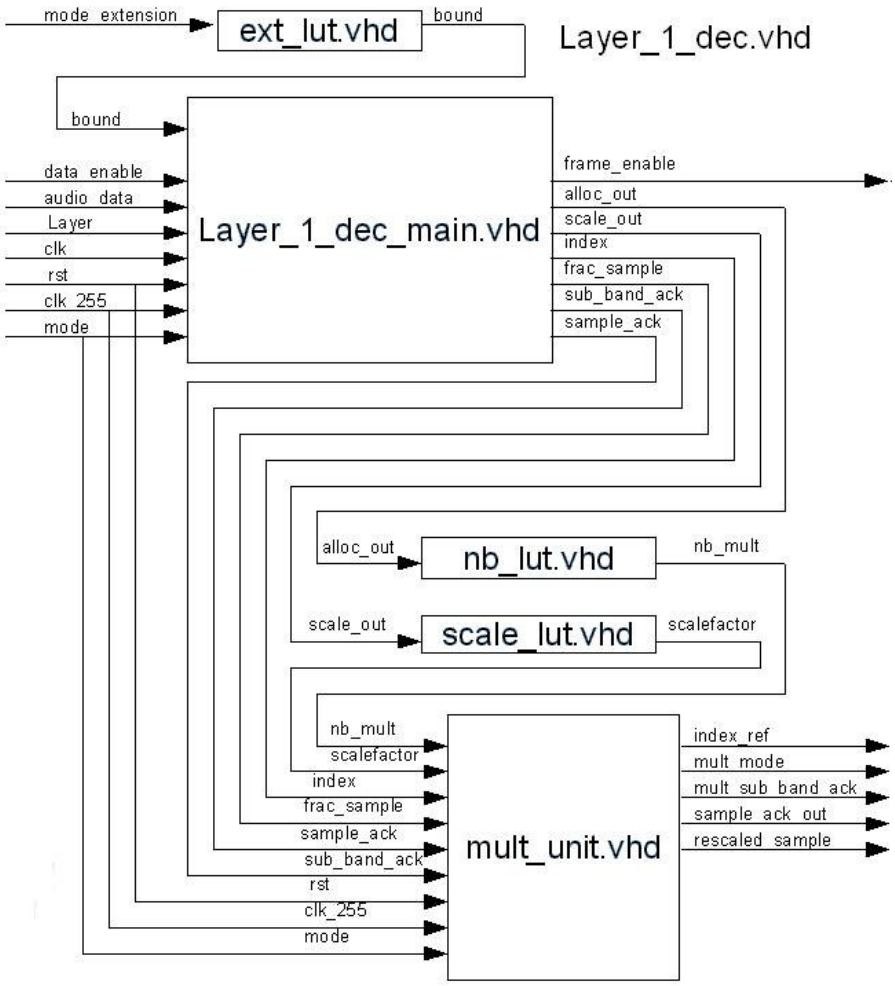


***Figure 29*** *Block diagram of Layer_1_dec.vhd*

### 3.10.1 Layer_1_decoder_main.vhd

The decompression of the audio sub-band samples takes place in the main part of the Layer I decoder (**Appendix 1-9**, `Layer_1_dec_main.vhd`) and is shown in Figure 30. A major part of `Layer_1_dec_main` is the case statements, in which there are four states as follows:

1. `proc_alloc_pro_Start` which is used to read the four bits of the `nb` of the sub-band samples;
2. `proc_scale_pro_Start` which is used to read the six bits of the scale factor of the sub-band samples;
3. `proc_sample_pro_Start` which is used to read the bits of 12 sub-band samples, the number of bits is the value of `nb`;
4. `proc_wait` which is used to wait until the next frame of compressed data is to be read.

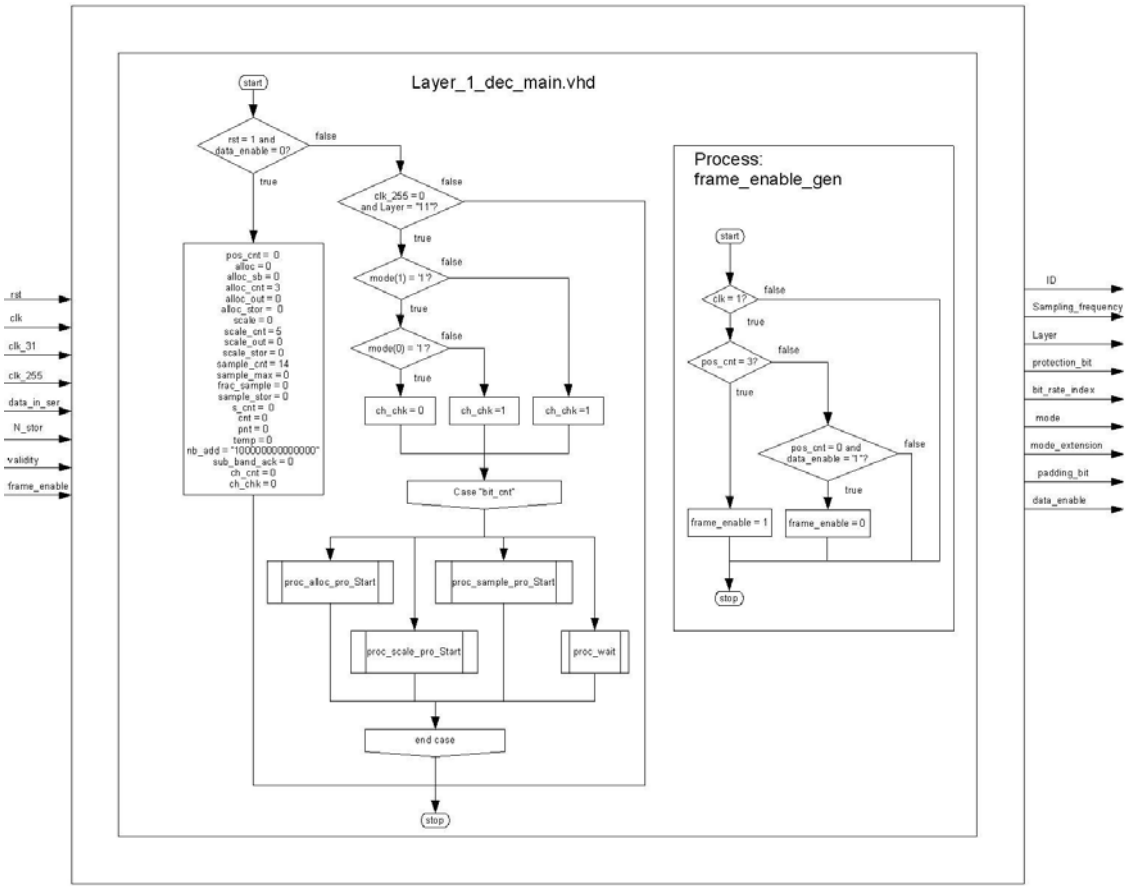


***Figure 30*** *Flowchart of Layer_1_main.vhd*

These procedures operate as follows:

**proc_alloc_pro_Start**

This procedure (Figure 31) reads in 32 four bit `nb` values from the compressed audio data section of the MPEG 1 Layer I frame and checks if the value is valid. If it isn't valid the `nb` is recorded as 0. If the bit allocation is valid an array `temp_counter` stores which of the 32 `nb` values this is, and the value of the `nb` is stored in the array `alloc` until 32 `nb` values have been read. For example, if the third `nb` is the only one valid, then the `temp_counter` array will store the value 3 in the first element, and the value of `nb` will be stored in the third element of the array `alloc`. Once the 32

values of nb have been read, the position count referred to as pos_cnt is set to 1 causing the case statement to exit to proc_scale_pro_Start.

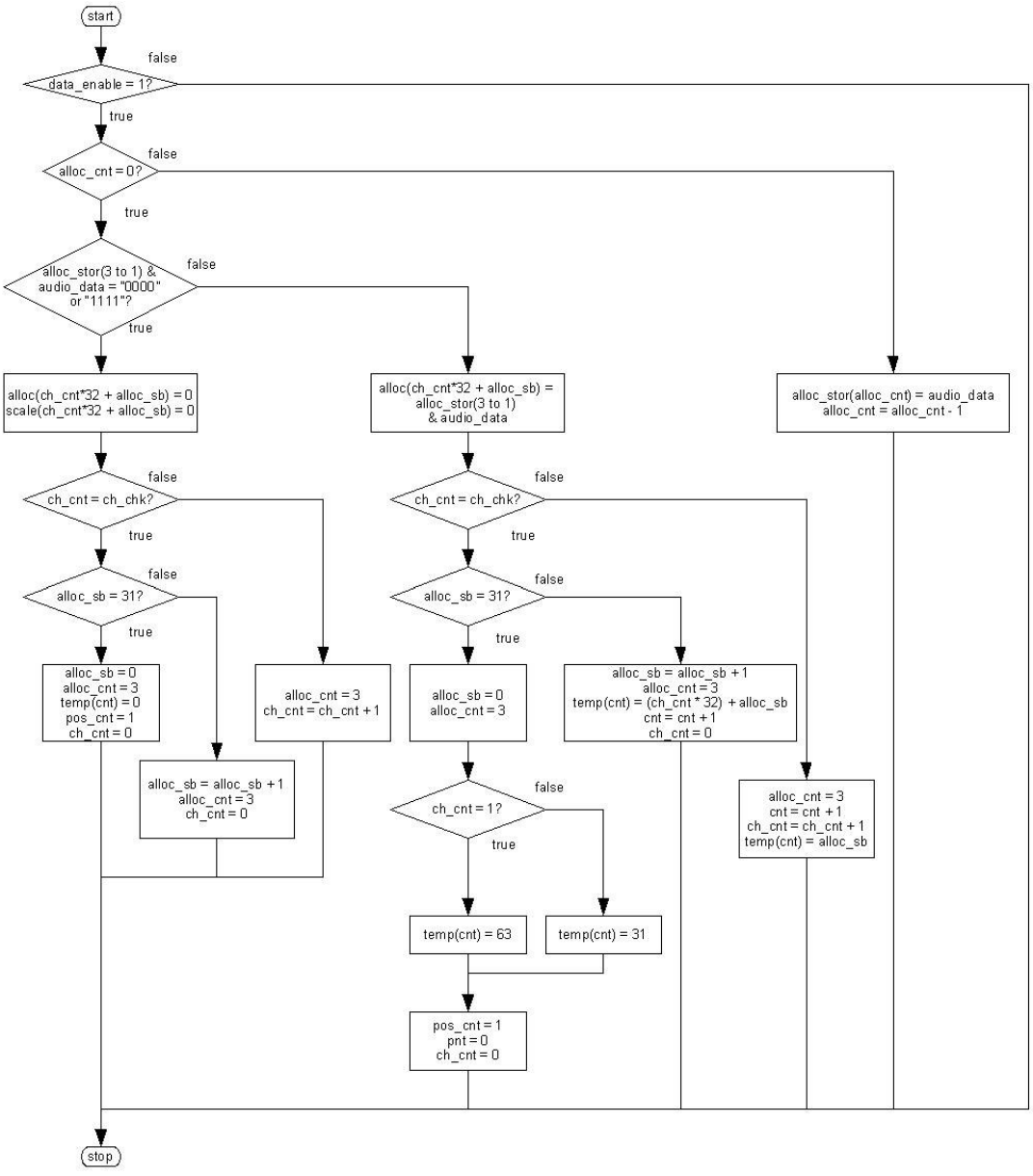


***Figure 31*** *Flowchart of procedure proc_alloc_pro_Start*

**proc_scale_pro_Start**

This procedure (Figure 32) reads in the six bits of a scale factor. The number of the valid nb that were read previously and stored in the array temp_counter are now

read back and act as the index to the array `scale` which stores the scale factors. Referring to the previous example, if the third `nb` was the only one valid, then the `temp_counter` array will read the value 3 from the first element, and then the only six bit scale factor read from the frame will be stored in the third element of the `scale` array. When all of the valid values for `nb` have had their corresponding scale factor read and allocated, the position count referred to as `pos_cnt` is incremented causing the case statement to exit.

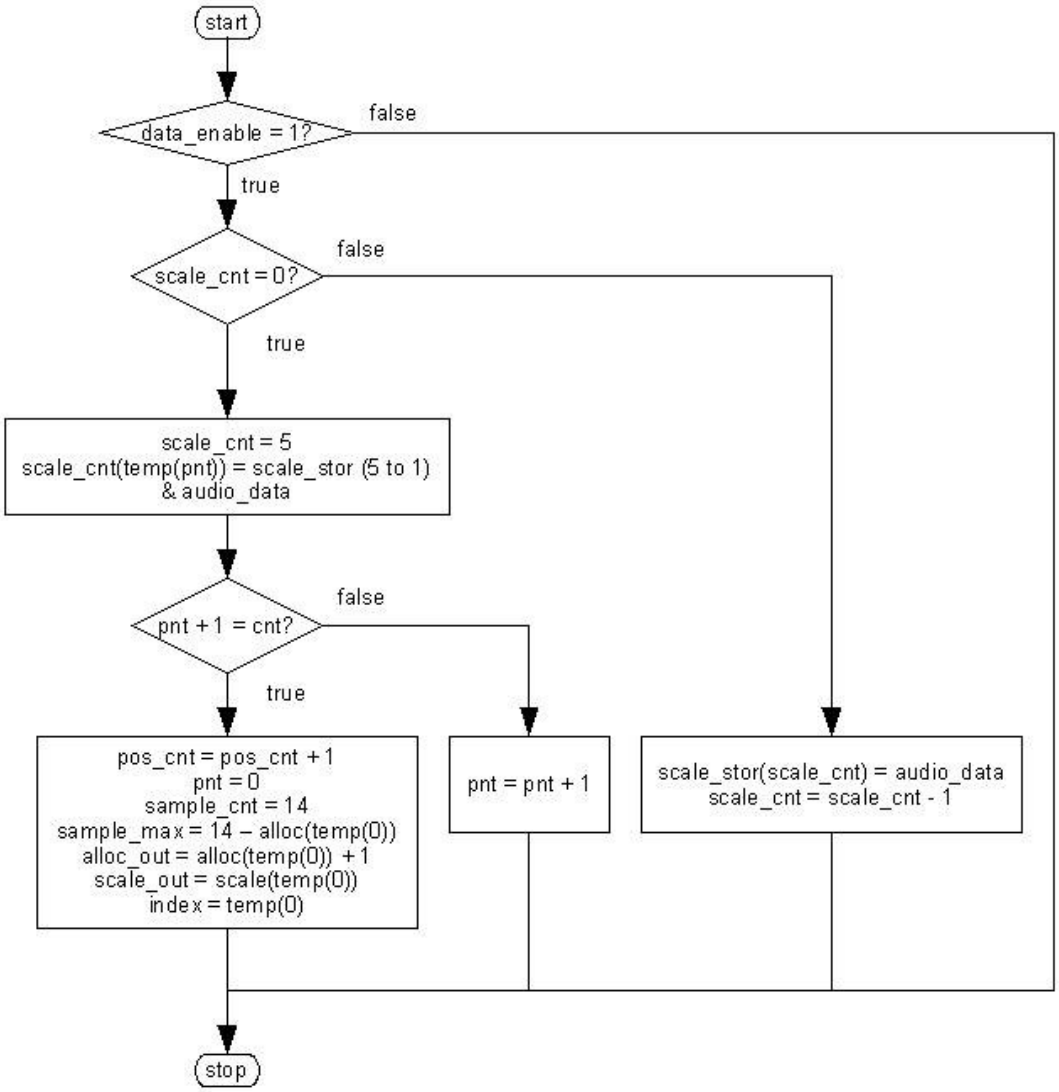


***Figure 32*** *Flowchart of procedure proc_scale_pro_Start*

**proc_sample_pro_Start**

This procedure (Figure 33) reads in the bits of the sub-band samples. The number of bits read for a sub-band sample was determined by the value of `nb` that was read in during the previous procedure. The actual `nb` values are read back from the array `alloc` to determine how many bits are read per sub-band sample. To continue the previous example, if the third bit allocation was the only valid bit allocation, then the `temp_counter` array will store the value 3 in the first element and then the only sub-band samples read from the frame for the next 12 samples will have the signal `index` set to 3. The 12 sub-band samples will have the number of bits in length specified by the value of bit allocation read from the third element of the array `alloc`. As it is read, the sub-band sample is stored in `sample_stor`. The first bit of the fourteen bits of a sub-band sample has to be inverted so that the sub-band sample has the correct polarity for the signed two's compliment format. When the last bit is read, and the first bit inverted, the constant `nb_add` (which is the value of $2^{(-nb+1)}$) must be added to the sub-band sample for the first part of requantisiation. When all of the valid bit allocations read have a corresponding scale factor read the position count referred to as `pos_cnt` is incremented causing the case statement to exit.

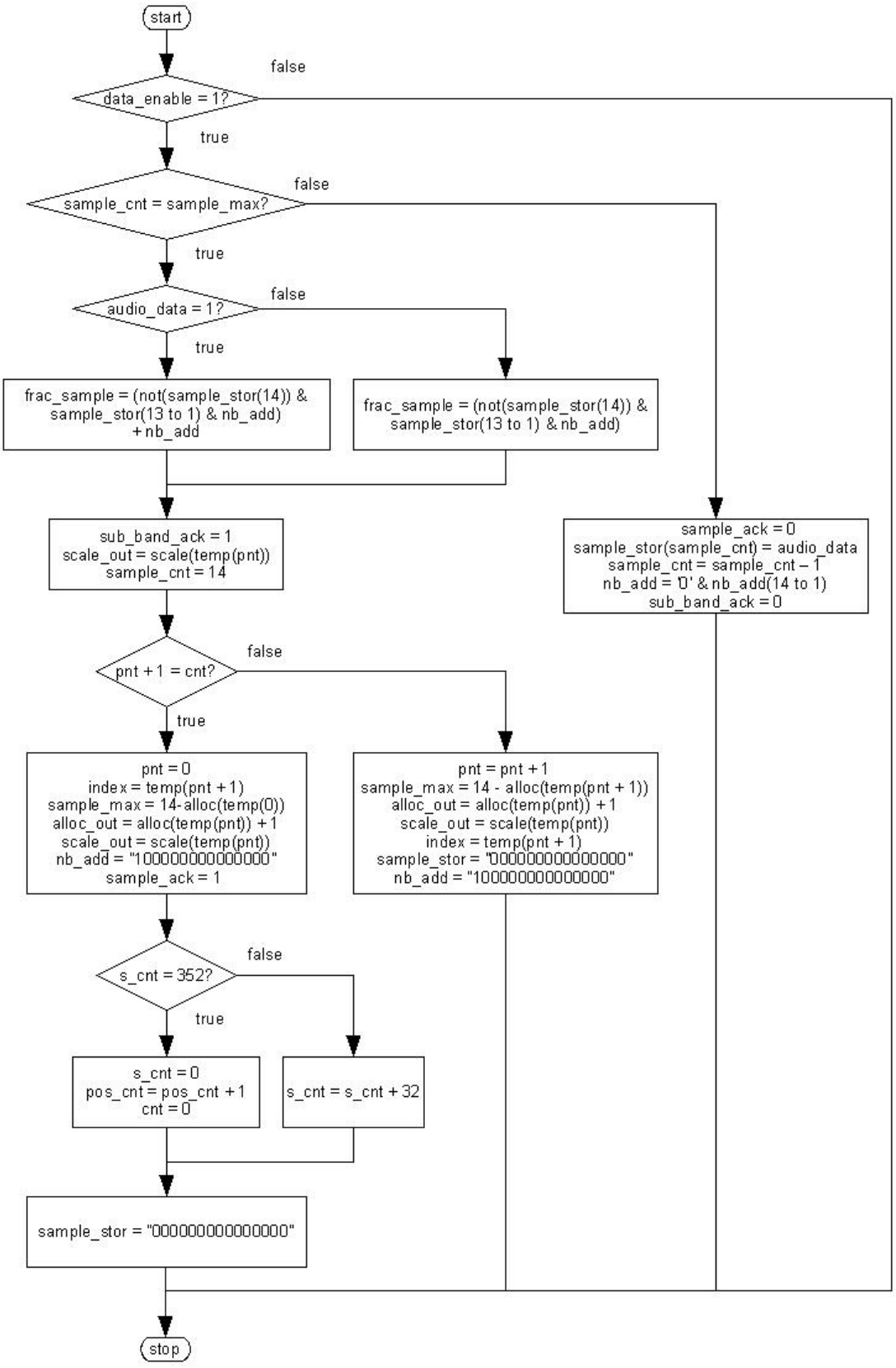


***Figure 33*** *Flowchart of procedure proc_sample_pro_Start*

**proc_wait**

This procedure resets all the counters and variables used in the other procedures;

### 3.10.2 Extension loop up table

When the compressed data is in joint stereo format, the "bound" limit (the number of sub-band samples that are common between both the left and right audio channels) is returned from this look up table (**Appendix 1-10**, `ext_lut.vhd`) when supplied with the mode extension as an index—i.e., the valid `nb`, the scale factors and the sub-band samples up to the "bound" limit are read once and then copied to the other channel. Because the `nb`, the scale factor and the sub-band sample are the same up to the "bound" limit, the data can be copied to the other channel. The four levels of bound specified in a MPEG audio frame as shown in Table 6.

### 3.10.3 Number of bits look up table

The `nb` per sub-band sample (referred to as `alloc_out`) is used as an index to this look up table (**Appendix 1-11**, `nb_lut.vhd`) to return the value of `nb_mult` (which is $2^{nb} / 2^{(nb-1)}$). This must be further multiplied by the sub-band sample for the second part of requantisiation.

### 3.10.4 Scalefactor look up table

This scale factor per sub-band (referred to as `scale_out`) is used as an index to this look up table (**Appendix 1-12**, `scale_lut.vhd`) to return the value of

`scalefactor`. This must then be multiplied by the sub-band sample for the third part of the requantisiation.

### 3.10.5 Multiply unit

The Multiply unit in the Layer I decoder of this model (shown in **Appendix 1-13**, `mult_unit.vhd` and shown in Figure 34) is passed the fractionalised compressed sub-band sample, the `nb_mult` (which is $2^{nb} / 2^{(nb-1)}$) and the `scalefactor`. These numbers are multiplied together to give the requantised sub-band sample which is in fact a restored two's complement fractional format binary number. The Multiply unit saves processing time by performing both multiplications at once.

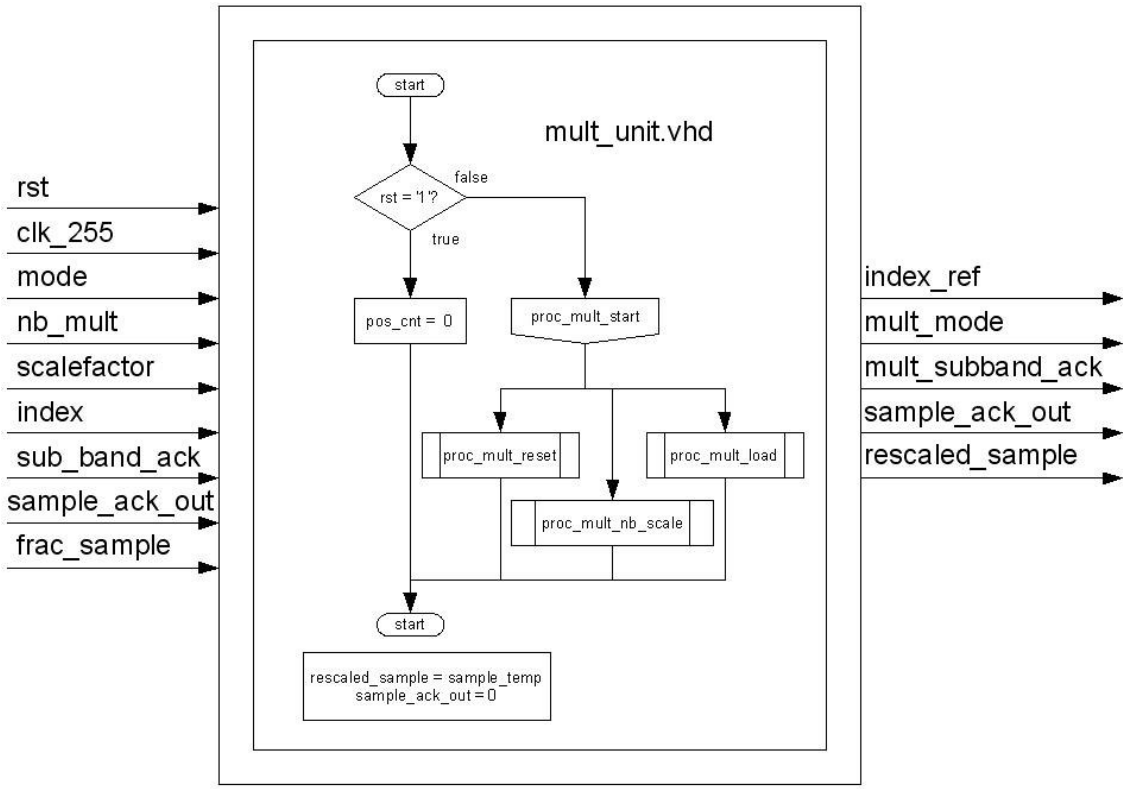


***Figure 34*** *Flowchart of mult_unit.vhd*

It can be seen that the Multiply unit consists of a case statement with three procedures as follows:

**proc_mult_reset**

This procedure resets all the counters and variables used in the other procedures.

**proc_mult_load**

This procedure waits for the data to be returned from the Number of bits and Scalefactor look up tables;

**proc_mult_nb_scale**

This procedure (Figure 35) multiplies the sub-band sample by the `nb_mult` and the `scalefactor` to give the requantised sub-band sample which is in fact a restored two's complement fractional format binary number.

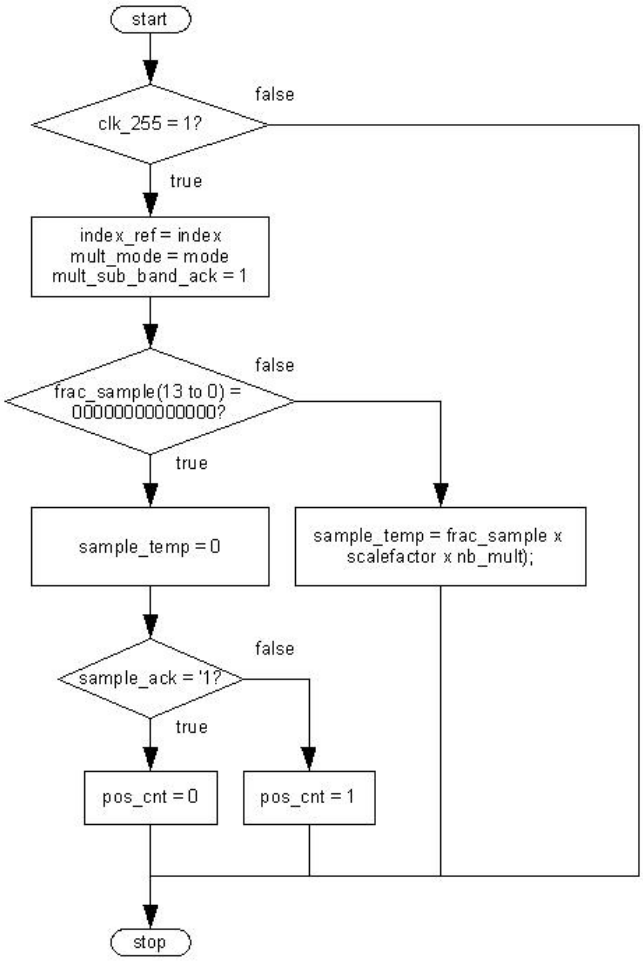


***Figure 35*** *Flowchart of proc_mult_nb_scale*

## 3.11 The Requantisation of Compressed Audio Sub-band Samples

As identified in **Section 2.10**, the compressed sample is restored to a fractional number using (4). It can be seen that this formula can be broken into two separate parts. The offset $2^{(-nb+1)}$ is added to an unrestored two's complement fractional format binary number, and the result is multiplied by $2^{nb} / 2^{(nb-1)}$. This results in a restored two's complement fractional format binary number.

| nb | -nb+1 | $2^{(-nb+1)}$ | Binary fractional format |
|---|---|---|---|
| 2 | -1 | 0.5000000000000000 | 010000000000000 |
| 3 | -2 | 0.2500000000000000 | 001000000000000 |
| 4 | -3 | 0.1250000000000000 | 000100000000000 |
| 5 | -4 | 0.0625000000000000 | 000010000000000 |
| 6 | -5 | 0.0312500000000000 | 000001000000000 |
| 7 | -6 | 0.0156250000000000 | 000000100000000 |
| 8 | -7 | 0.0078125000000000 | 000000010000000 |
| 9 | -8 | 0.0039062500000000 | 000000001000000 |
| 10 | -9 | 0.0019531250000000 | 000000000100000 |
| 11 | -10 | 0.0009765625000000 | 000000000010000 |
| 12 | -11 | 0.0004882812500000 | 000000000001000 |
| 13 | -12 | 0.0002441406250000 | 000000000000100 |
| 14 | -13 | 0.0001220703125000 | 000000000000010 |
| 15 | -14 | 0.0000610351562500 | 000000000000001 |

***Table 11*** *nb_add constants look up table*

As shown in Table 9, `nb` can only take on values of 0 and between 2 and 15. As a bit allocation of 0 means that no bits are allocated to a specific sub-band sample, there is no need for further processing of this sample. This means that there is in fact only 14 possible values are shown in Table 11. A binary fractional format number is used to represent the offset value as shown in the right-most column of Table 11. This binary

fractional format allows the 14 values to be calculated to an initial value of 010000000000000 applying followed by a number of right shifts. For example, if the number of bits allocated to each sample in the sub-band is three, 010000000000000 shifted right three times to give 000010000000000. Thus, the 15 values of the offset do not have to be stored in memory, but can be implemented as a simple shift operation.

The offset is called `nb_add` in the VHDL code and is implemented by having first initialised it to 100000000000000, which is then right shifted within the loop that reads in the number of bits using VHDL code shown in (12) from **Appendix 1-9**, `Layer_1_dec_main.vhd`.

$$nb\_add <= '0' \& nb\_add(14 \text{ downto } 1) \tag{12}$$

When the number of bits specified by the bit allocation has been counted the final value of `nb_add`, is added to the unrestored two's complement fractional format binary number using (13) and stored in the variable `frac_sample`.

$$frac\_sample = s'' + nb\_add = s'' + 2^{(-nb+1)} \tag{13}$$

The second part of the function in (4) is reproduced in (14), is also dependent on `nb` and is given the name `nb_mult` in the VHDL code. As before, a bit allocation of 0 means that no bits are allocated to a specific sub-band sample, so there are also only 14 values that can be taken on by `nb_mult`.

$$nb\_mult = \frac{2^{nb}}{2^{nb} - 1} \tag{14}$$

The values generated by (14) and converted to a binary fractional format are shown in the right-most column of Table 12.

| nb | $2^{nb}$ | $2^{nb-1}$ | $(2^{nb})/(2^{nb-1})$ | Binary fractional format |
|---|---|---|---|---|
| 2 | 4 | 3 | 1.33333333333333 | x"15555555555555" |
| 3 | 8 | 7 | 1.14285714285714 | x"12492492492492" |
| 4 | 16 | 15 | 1.06666666666667 | x"11111111111111" |
| 5 | 32 | 31 | 1.03225806451613 | x"10842108421084" |
| 6 | 64 | 63 | 1.01587301587302 | x"10410410410410" |
| 7 | 128 | 127 | 1.00787401574803 | x"10204081020408" |
| 8 | 256 | 255 | 1.00392156862745 | x"10101010101010" |
| 9 | 512 | 511 | 1.00195694716243 | x"10080402010080" |
| 10 | 1024 | 1023 | 1.00097751710655 | x"10040100401004" |
| 11 | 2048 | 2047 | 1.00048851978505 | x"10020040080100" |
| 12 | 4096 | 4095 | 1.00024420024420 | x"10010010010010" |
| 13 | 8192 | 8191 | 1.00012208521548 | x"10008004002001" |
| 14 | 16384 | 16383 | 1.00006103888177 | x"10004001000400" |
| 15 | 32768 | 32767 | 1.00003051850948 | x"10002000400080" |

***Table 12*** *nb_mult constants look up table*

Table 12 is implemented using a look up table called `nb_lut.vhd` (as discussed previously in **Section 3.10.3**) which forms an input to the multiplier unit. When the multiplier unit requires the value of `nb_mult`, the look up table `nb_lut.vhd`, is given the number of bits by the variable `alloc_out`, and the value of `nb_mult` is returned. The variable `nb_mult` is multiplied by `frac_sample` to give `nb_mult_sample` which is the restored two's complement fractional format binary

number represented by *s''*. This is shown in (15) where (13) and (14) have been substituted in (4).

$$\begin{aligned} s'' &= nb\_mult\_sample = \frac{2^{nb}}{2^{nb}-1} \times (s''' + 2^{(-nb+1)}) \\ s'' &= nb\_mult\_sample = \frac{2^{nb}}{2^{nb}-1} \times (s''' + nb\_add) \\ s'' &= nb\_mult\_sample = nb\_mult \times frac\_sample \end{aligned} \tag{15}$$

This restored sub-band sample then has to be rescaled, and this is achieved by multiplying the restored two's complement fractional format binary number by a scale factor as shown in (16). The value of the scale factor is determined by the scalefactor index read before and used as an index for Table 10. After this multiplication the sub-band sample is said to be a rescaled sub-band sample represented by *s'*.

$$s' = scalefactor \times s'' \tag{16}$$

At this point a compressed audio sample sub-band is said to be *requantised* and is passed on to the polyphase synthesis sub-band filter in order to calculate 32 decompressed audio samples.

## 3.12 Polyphase Synthesis Sub-band Filter

The Polyphase Synthesis Sub-band Filter for an MPEG 1 Layer I is depicted in Figure 36. The implementation in VHDL of this filter is very different compared to the one described in the standard [1]. Although it performs the same mathematical operations,

it does so in a different order to achieve an increase in speed whilst using a smaller amount of resources. Instead of implementing each step as shown in Figure 36 as separate processes, the VHDL model combines several these processes, which results in two major processing blocks and two look up tables as shown in Figure 37. The two major parts are the Nik controller (`Nik_cont.vhd`) and the Di controller (`Di_cont.vhd`) and the two look up tables are the Nik look up table (`Nik_lut.vhd`) and the Di look up table (`Di_lut.vhd`). Each of these look up tables contain the filter coefficients used for the matrixing and widowing functions respectively, as shown in Figure 36.

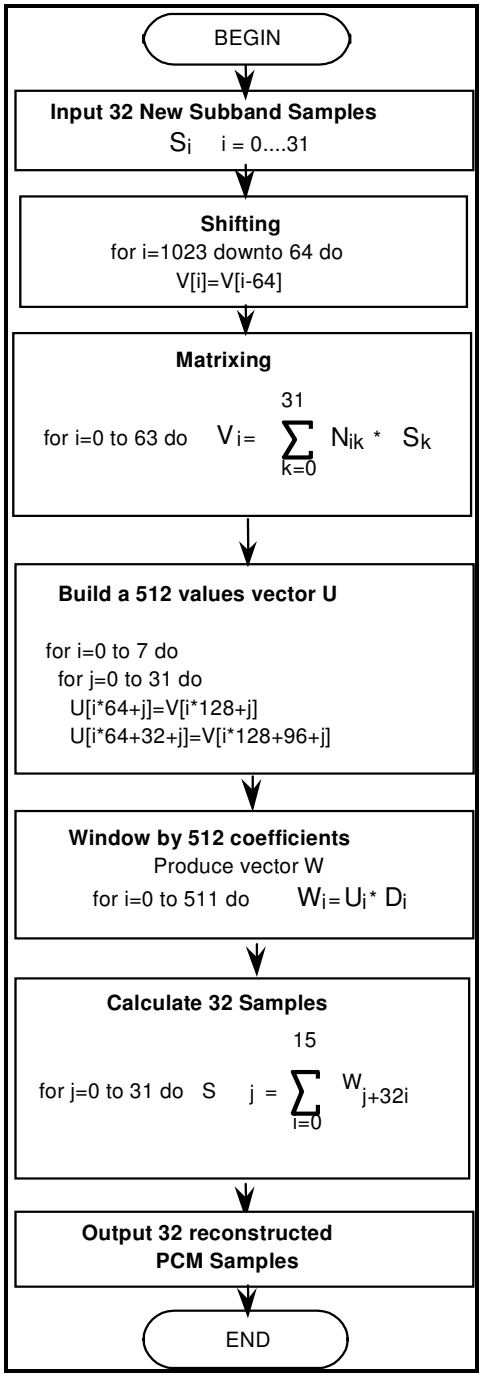


***Figure 36*** *Flowchart of Synthesis sub-band filter flow chart*

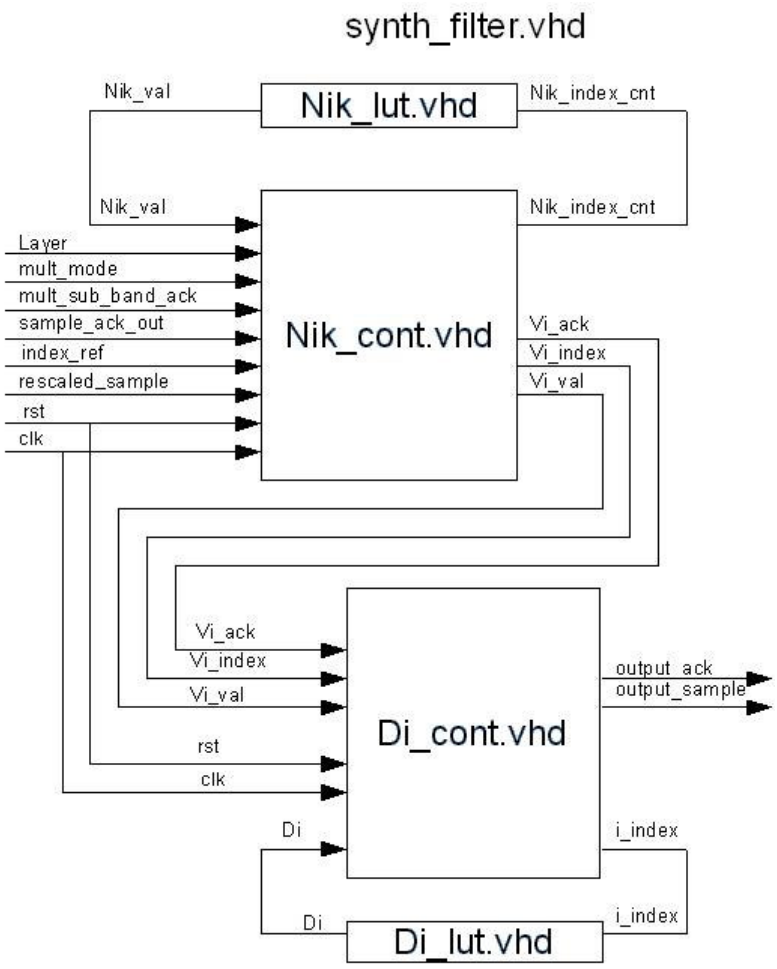


***Figure 37*** *Flowchart synth_filter.vhd*

The Nik controller encompasses the first and third processes shown in Figure 36, while the Di controller contains the rest of the processes. The Polyphase Synthesis Sub-band Filter, implemented according to the standard, requires the resources of a pair of 32 element arrays, a single 64 element array, a pair of 512 element arrays and a single 1024 element array. In comparison the VHDL model uses an implementation of the Polyphase Synthesis Sub-band Filter that uses only a pair of 32 element arrays, a single 64 element array and a single 1024 element array.

The VHDL code of the Polyphase Synthesis Sub-band Filter is made up of the top layer `synth_filter.vhd` shown in Figure 37, plus `Nik_cont.vhd` and `Di_cont.vhd` for the Nik controller and the Di controller, `Nik_lut.vhd` and `Di_lut.vhd` for the Nik look up table and the Di look up table, and `Nik_rom_def.vhd` and `Di_rom_def.vhd` for the data used by the Nik look up table and the Di look up table. The listing of these VHDL files can be found in the **Appendix 1-14**.

### 3.12.1 The Nik Controller

The Nik controller (Figure 38) is the first part of the VHDL model implementation of the Polyphase Synthesis Sub-band Filter and contains the first and third processes of the Polyphase Synthesis Sub-band Filter of the MPEG 1 Layer I audio decoder shown in Figure 36 (`Input 32 New Subband Samples` and `Matrixing`). The Nik controller receives its data from the Layer I decoder in the form of 32 sub-band samples and after processing outputs 64 matrixed values to the Di controller.

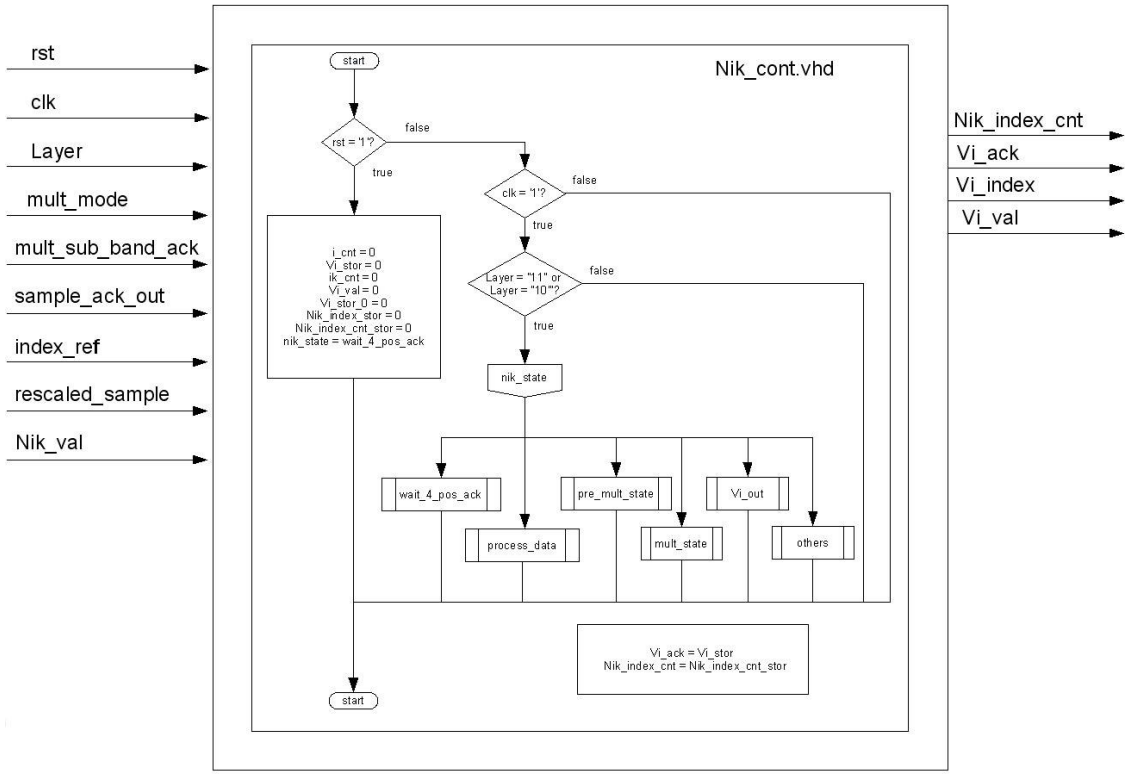


***Figure 38*** *Flowchart of Nik_cont.vhd*

The Nik controller combines the first and third processes into one process by loading a sub-band sample from the Layer I decoder, and the corresponding matrixing coefficient from the Nik look up table, and performing the mathematical calculations required by the matrixing that use this sub-band sample. The Matrixing is as specified by the standard [1] using (17).

**for i = 0 to 63**

$$V(i) = \sum_{k=0}^{31} (N_{ik} \times S_k) \qquad (17)$$

where $S_k$ are the 32 sub-band samples, $N_{ik}$ are the matrixing coefficients of the Polyphase Synthesis Sub-band Filter, $i$ is the index for the 64 matrixed values and $k$ is the index to the 32 sub-band samples.

This can be rewritten in the expanded form.

$$
\begin{aligned}
&V(0) = (N_{0,0} \times S_0) + (N_{0,1} \times S_1) + (N_{0,2} \times S_2) + .. + (N_{0,31} \times S_{31}) \\
&V(1) = (N_{1,0} \times S_0) + (N_{1,1} \times S_1) + (N_{1,2} \times S_2) + .. + (N_{1,31} \times S_{31}) \\
&V(2) = (N_{2,0} \times S_0) + (N_{2,1} \times S_1) + (N_{2,2} \times S_2) + .. + (N_{2,31} \times S_{31}) \\
&.. \\
&.. \\
&.. \\
&.. \\
&V(63) = (N_{63,0} \times S_0) + (N_{63,1} \times S_1) + (N_{63,2} \times S_2) + .. + (N_{63,31} \times S_{31})
\end{aligned}
$$

As can been seen the function can be expanded into a series of multiply/accumulate operations to calculate the 64 matrixed elements that are to be stored in the lower 64 positions of the 1024 element matrix array. Each of these operations has two input variables, the sub-band sample and a matrixing coefficient. Each sub-band sample is used once for a multiply and accumulate mathematical operation for each of the 64 matrixed elements. That is to say, only the first multiply and accumulate operation of each row requires $S_0$, only the second multiply and accumulate operation of each row requires $S_1$ etc... Thus in the standard mechanisms, all 32 requantised compressed audio samples would have to be received before calculating any of the matrixed elements.

The Nik controller of this VHDL model performs calculations column by column, instead of calculating row by row, meaning that the multiply and accumulate operations for each of the 64 matrixed elements can be performed as soon as the corresponding sub-band sample is received. This means that the time spent waiting for the decoder to process the next sub-band sample is used to perform all mathematical operations on the previous requantised compressed audio sample. Once the 32 sub-band samples have

been received, and the multiply and accumulate mathematical operations for each of the 64 matrixed elements for all 64 rows have been calculated, the results are written to the lower 64 elements of the 1204 element matrix array.

The VHDL code of the Nik controller (Figure 38) is shown in **Appendix 1-15** and called `Nik_cont.vhd`. The code in `synth_filter.vhd` passes the `nb_mult_sample_out` variable from `Layer_1_dec` and the Nik filter coefficient (`Nik_val`) from the Nik look up table. This variable `nb_mult_sample_out` contains the value of a requantised sub-band sample. The VHDL code of the Nik controller is a state machine made up of five procedures as follows:

1. `wait_4_pos_ack` (Figure 39) which waits for the rising edge of the variable `mult_sub_band_ack` to signify that the multiplier unit in the Layer 1 decoder has finished multiplying and has a new requantised sub-band samples for the Nik controller before moving to the `process_data` procedure;

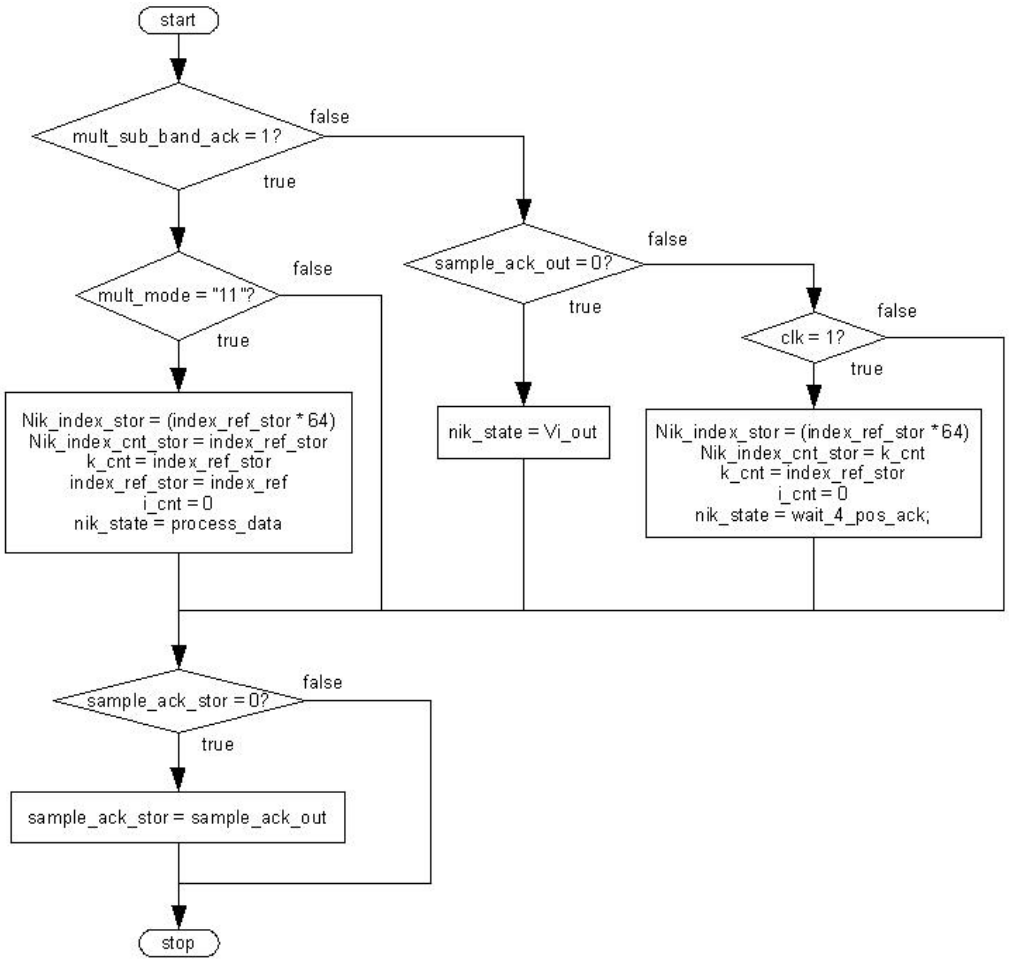


***Figure 39*** *Flowchart of wait_4_pos_ack*

2. `process_data` (Figure 40) where the variable `Vi_ack` is set high to signify to the Di controller that the Nik controller is starting to process the next requantisied sub-band sample before moving to the `pre_mult_state` procedure. Note here that, unlike the remaining states, this is active on the negative edge of the clock. This corrects a timing issue that would otherwise cause the first value to be skipped—by changing to negative edge triggered, the signals are guaranteed to be stable for the first value;

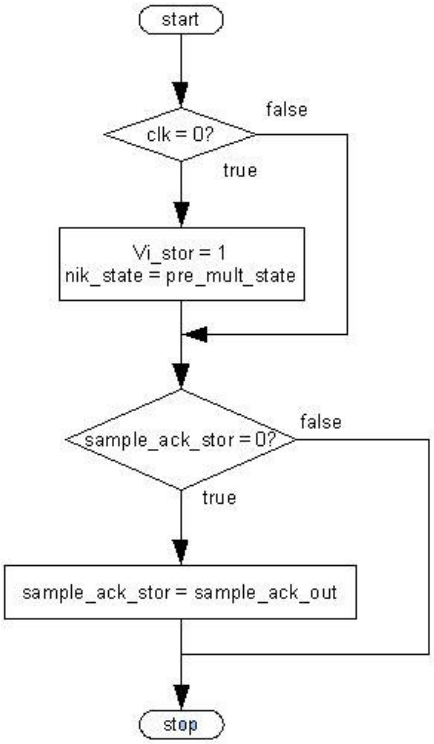


***Figure 40*** *Flowchart of process_data*

3. `pre_mult_state` (Figure 41) to have the Nik controller read the current value of the sum of previous multiply and accumulates of the requantisied sub-band sample by Nik filter coefficients from the array `Vi_stor_0` before advancing to the procedure `mult_state`;

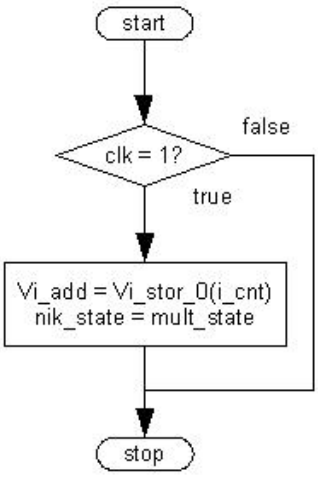


***Figure 41*** *Flowchart of pre_mult_state*

4. `mult_state` (Figure 42) performs a multiply and accumlate whereby the requantisied sub-band sample is multiplied by the Nik filter coefficient supplied from the Nik look up table and added to the sum of previous multiply of the requantisied sub-band sample by Nik filter coefficients;

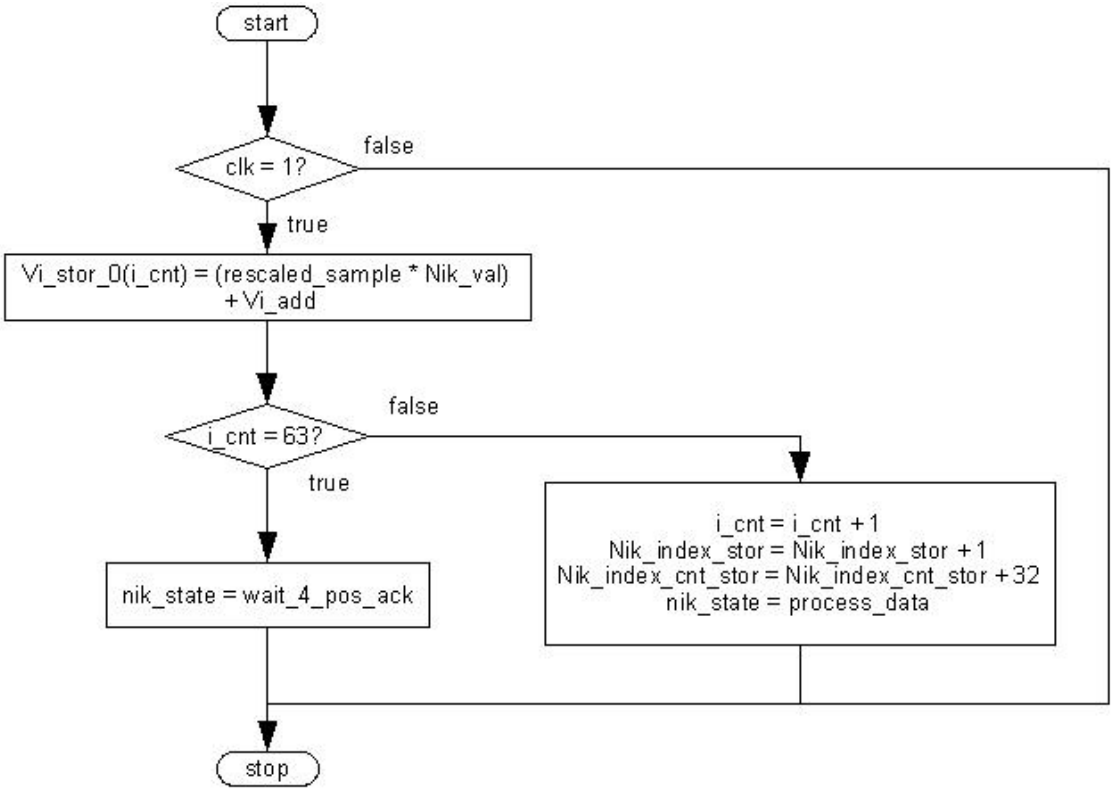


***Figure 42*** *Flowchart of mult_state*

5. `Vi_out` (Figure 43) which outputs all 64 newly calculated values of `Vi` to the Di controller and returns to the `wait_4_pos_ack` procedure to wait for the next requantisied sub-band sample;

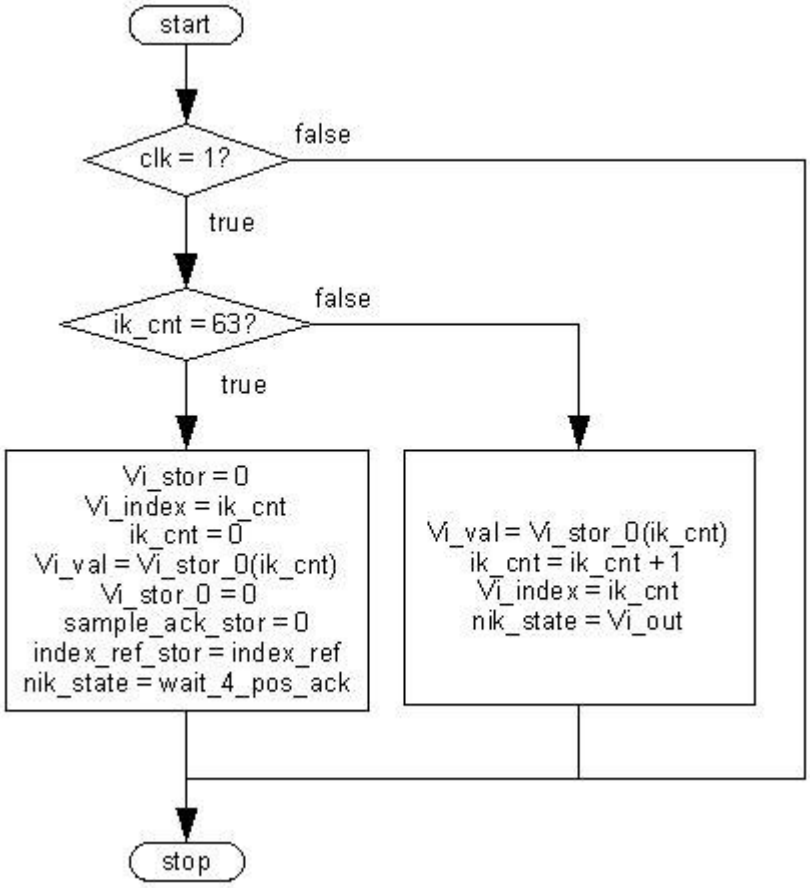


***Figure 43*** *Flowchart of Vi_out*

### 3.12.2 The Nik look up table

The Nik look up table is described using two files of VHDL code listed in the **Appendix**, as follows:

- Nik look up table shown in **Appendix 1-16** and called `Nik_lut.vhd`;
- Nik definition file shown in **Appendix 1-17** and called `Nik_rom_def.vhd`;

The Nik look up table receives an index value from the Nik controller. This index value indicates which Nik filter coefficient is required for the next multiply operation.

The Nik look up table indexes the Nik definition file and returns the required Nik filter coefficient.

The Nik definition file stores the 2048 Nik filter coefficient values that are calculated as follows:

**for i = 0 to 63**
**& k = 0 to 31**

$$N_{ik} = \cos[(16+i)(2k+1)(\pi/64)] \quad (18)$$

where $N_{ik}$ are the matrixing coefficients of the Polyphase Synthesis Sub-band Filter, $i$ is the index to the lower 64 elements of the matrixing coefficients of the Polyphase Synthesis Sub-band Filter, $k$ is the index to the 32 sub-band samples and $\pi$ is the mathematical constant.

The $N_{ik}$ matrixing coefficients of the Polyphase Synthesis Sub-band Filter are not dependent on the sub-band samples, therefore they can be precalculated. These precalculated matrixing coefficients were then converted to a signed decimal binary representation.

### 3.12.3 The Di Controller

The Di controller is the second part of the VHDL model implementation of the Polyphase Synthesis Sub-band Filter. It receives 64 matrixed values from the Nik controller and stores these values in a 1024 element array. These values are then read in a specific order, set by a specialised counter, to reduce the 1024 elements to 512.

These resultant multiplied values are then accumulated in order to convert the sub-band samples into a decoded audio sample. This is repeated until 32 decoded audio samples have been calculated, at which time the Di controller waits for the next 64 matrixed values from the Nik controller. The specialised counter removes the need for a separate 512 element array by combing the windowing operation seen in the fourth process of the Polyphase Synthesis Sub-band Filter shown in Figure 36 with second, fifth, sixth and seventh processes.

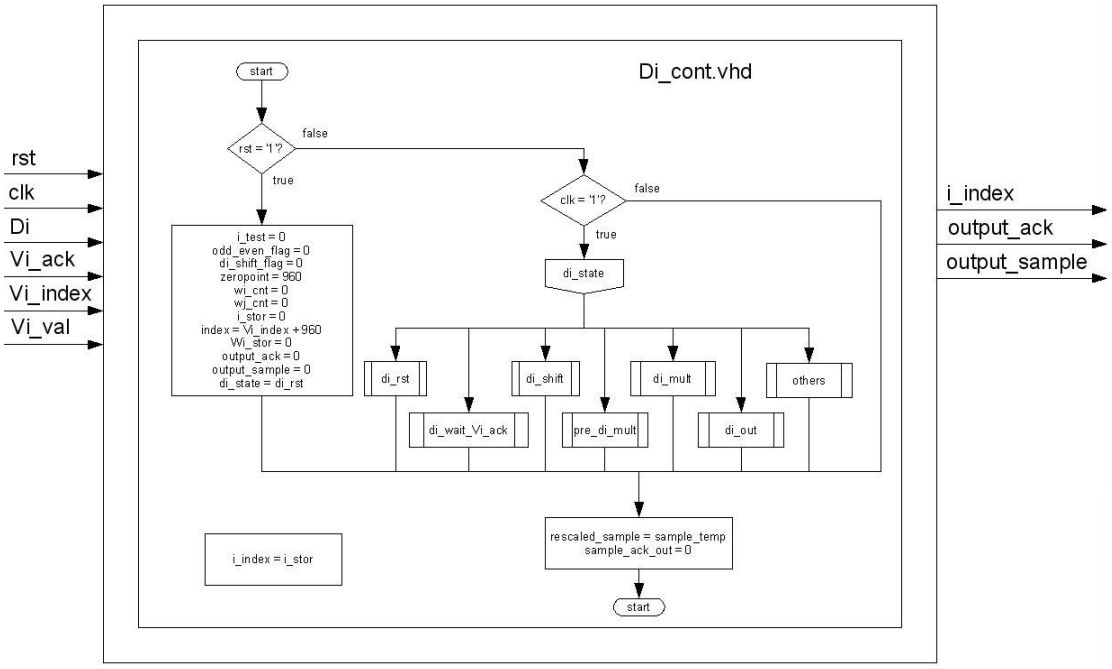


***Figure 44*** *Flowchart of Di controller*

The second process of the Polyphase Synthesis Sub-band Filter of the MPEG 1 Layer I audio decoder shown in Figure 36, is the *Shifting* process. This process shifts the lower 960 elements of the 1024 element matrix array, to the upper 960 elements of the matrix array by applying (19).

**for** $i$ **= 1024 down to 64**

$$V[i] = V[i-64] \tag{19}$$

where *V(i)* are the 1024 elements of the array and *i* is the index to the array of *V*.

The Di controller does not shift data as shown in (19). Instead the Di controller uses a counter that represents the zero position of the array, and decrements its value by 64 positions each time 64 matrixed values, represented by *Vi*, are received from the Nik controller. When the counter gets to 0, on the next shift process the counter wraps and decrements down from 1023 to give 960. This means that no time is spent moving data from element of the matrix array to another. However this means that for each read or write, to or from, a position in the matrix array specified by an index, the counter that represents the zero position of the array has to be added to the index used.

The sixth process of the Polyphase Synthesis Sub-band Filter shown in Figure 36 is to calculate 32 samples of decompressed audio data using:

**for j = 0 to 31**

$$S_j = \sum_{i=0}^{15} W_{j+32i} \tag{20}$$

where *S* is the 32 decompressed audio samples, *W* is the 512 value array of windowed data, and *i* and *j* are the indexes to the windowed data and the decompressed audio samples.

This can be expanded and rewritten in the following form:

$$\begin{aligned}
S_0 &= W_{0+32\times0} + W_{0+32\times1} + W_{0+32\times2} + W_{0+32\times3} + W_{0+32\times4} + W_{0+32\times5} + W_{0+32\times6} + .. + W_{0+32\times15} \\
S_1 &= W_{1+32\times0} + W_{1+32\times1} + W_{1+32\times2} + W_{1+32\times3} + W_{1+32\times4} + W_{1+32\times5} + W_{1+32\times6} + .. + W_{1+32\times15} \\
S_2 &= W_{2+32\times0} + W_{2+32\times1} + W_{2+32\times2} + W_{2+32\times3} + W_{2+32\times4} + W_{2+32\times5} + W_{2+32\times6} + .. + W_{2+32\times15} \\
&.. \\
&.. \\
&.. \\
S_{31} &= W_{31+32\times0} + W_{31+32\times1} + W_{31+32\times2} + W_{31+32\times3} + W_{31+32\times4} + W_{31+32\times5} + W_{31+32\times6} + .. + W_{31+32\times15}
\end{aligned}$$

Which can then be simplified and rewritten as follows:

$$
\begin{aligned}
S_0 &= W_{0+0} + W_{0+32} + W_{0+64} + W_{0+96} + W_{0+128} + W_{0+160} + W_{0+192} + .. + W_{0+480} \\
S_1 &= W_{1+0} + W_{1+32} + W_{1+64} + W_{1+96} + W_{1+128} + W_{1+160} + W_{1+192} + .. + W_{1+480} \\
S_2 &= W_{2+0} + W_{2+32} + W_{2+64} + W_{2+96} + W_{2+128} + W_{2+160} + W_{2+192} + .. + W_{2+480} \\
&.. \\
&.. \\
&.. \\
S_{31} &= W_{31+0} + W_{31+32} + W_{31+64} + W_{31+96} + W_{31+128} + W_{31+160} + W_{31+192} + .. + W_{31+480}
\end{aligned}
$$

This can then be further simplified and rewritten as:

$$
\begin{aligned}
S_0 &= W_0 + W_{32} + W_{64} + W_{96} + W_{128} + W_{160} + W_{192} + .. + W_{480} \\
S_1 &= W_1 + W_{33} + W_{65} + W_{97} + W_{129} + W_{161} + W_{193} + .. + W_{481} \\
S_2 &= W_2 + W_{34} + W_{66} + W_{98} + W_{130} + W_{162} + W_{194} + .. + W_{482} \\
&.. \\
&.. \\
&.. \\
S_{31} &= W_{31} + W_{63} + W_{95} + W_{127} + W_{159} + W_{191} + W_{223} + .. + W_{511}
\end{aligned}
$$

So finally, the 32 decompressed audio samples may be calculated by starting at each of the first 32 elements of the window array, and adding every 32$^{nd}$ element of the windowed array *W*. For example, at sample 0, represented by $S_0$, add the element 0 of the windowed data array (represented by $W_0$) to the element 32 of the windowed data array, represented by $W_{32}$, and so on followed to element 64.

The fifth process of the Polyphase Synthesis Sub-band Filter of the MPEG 1 Layer I audio decoder shown (Figure 36) is the windowing operation where by the 512 values that make up the vector array *U*, are multiplied by the windowing coefficients to give the of *W*. This mathematical operation is shown in (21). It can be seen that each of the window array elements represented by $W_i$, is equal to the element with the same index of the vector array represented by $U_i$, multiplied by the filter coefficients with the same

index. These filter coefficients are given in the ISO/IEC 11172 standard [1] and represented by $D_i$.

**for i = 0 to 511**

$$Wi = Ui \times Di \tag{21}$$

where $W$ is the 512 value array of windowed data, $U$ is the 512 value array of vectored data, $D_i$ are the filter coefficients that are given in the ISO/IEC 11172 standard [1] and $i$ is an index to the windowed data, the vectored data and the filter coefficients.

The rewritten form of (20) can have each element of the windowed array represented by $W$, substituted for the vectored data represented by $U$, multiplied by the filter coefficients represented by $D$, as shown in (21) to give the form shown below. This substitution means that the VHDL model of the filter does not need to build a 512 value array to store the results of the multiplication shown in (20).

$$\begin{aligned}
S_0 &= (U_0 \times D_0) + (U_{32} \times D_{32}) + (U_{64} \times D_{64}) + (U_{96} \times D_{96}) + (U_{128} \times D_{128}) + .. + (U_{480} \times D_{480}) \\
S_1 &= (U_1 \times D_1) + (U_{33} \times D_{33}) + (U_{65} \times D_{65}) + (U_{97} \times D_{97}) + (U_{129} \times D_{129}) + .. + (U_{481} \times D_{481}) \\
S_2 &= (U_2 \times D_2) + (U_{34} \times D_{34}) + (U_{66} \times D_{66}) + (U_{98} \times D_{98}) + (U_{130} \times D_{130}) + .. + (U_{482} \times D_{482}) \\
&.. \\
S_{31} &= (U_{31} \times D_{31}) + (U_{63} \times D_{63}) + (U_{95} \times D_{95}) + (U_{127} \times D_{127}) + (U_{159} \times D_{159}) + .. + (U_{511} \times D_{511})
\end{aligned}$$

The fourth process of the Sub-band Filter reduces the 1024 elements of the $V$ array down to the 512 values of the $U$ array by applying (22).

**for i = 0 to 7**
**for j = 0 to 31**

$$\begin{aligned}
U(i \times 64 + j) &= V(i \times 128 + j) \\
U(i \times 64 + 32 + j) &= V(i \times 128 + 96 + j)
\end{aligned} \tag{22}$$

This represents a simple linear transformation of $V$ into $U$ and thus results of (22) can be calculated in advance. When calculated and reordered in the sequence required by

rewritten form of (21), it becomes obvious that these results of the (22) could be derived by the use of a count sequence that acts as an index to the array *V*. For the first output sample to be calculated the count sequence starts at 0, after this the count sequence increments by 96, then by 32. This is repeated for 16 values of the index to *V*. The use of this counter means that the VHDL model of the filter does not need to build a 512 value vector array with the data from the matrix array.

The rewritten form of (21) with the value of *U* substituted with the relevant value of *V* is shown below.

$$
\begin{aligned}
S_0 &= (V_0 \times D_0) + (V_{96} \times D_{32}) + (V_{128} \times D_{64}) + (V_{224} \times D_{96}) + (V_{256} \times D_{128}) + .. + (V_{992} \times D_{480}) \\
S_1 &= (V_1 \times D_1) + (V_{97} \times D_{33}) + (V_{129} \times D_{65}) + (V_{225} \times D_{97}) + (V_{257} \times D_{129}) + .. + (V_{993} \times D_{481}) \\
S_2 &= (V_2 \times D_2) + (V_{98} \times D_{34}) + (V_{130} \times D_{66}) + (V_{226} \times D_{98}) + (V_{258} \times D_{130}) + .. + (V_{994} \times D_{482}) \\
&.. \\
S_{31} &= (V_{31} \times D_{31}) + (V_{127} \times D_{63}) + (V_{159} \times D_{95}) + (V_{255} \times D_{127}) + (V_{287} \times D_{159}) + .. + (V_{1023} \times D_{511})
\end{aligned}
$$

The seventh and final process simply outputs the 32 reconstructed PCM samples are represented by the $S_k$ values.

The VHDL code of the Di controller (Figure 44) is shown in **Appendix 1-18** and called `Di_cont.vhd`. The code in `synth_filter.vhd` passes the variables `Vi_val` and `Vi_index` from the Nik controller (`Nik _cont.vhd`) and the Di filter coefficient (`Di`) from the Di look up table (`Di_lut.vhd`). The variable `Vi_index` informs the Di controller as to which `Vi_value` is being passed. The `Vi_val` is one

of the 64 matrixed values calculated from 32 requantised sub-band samples. The VHDL code of the Di controller is a state machine of six procedures:

1. `di_rst` (Figure 45) resets the index and zeropoint before advancing to the procedure `di_wait_Vi_ack`;

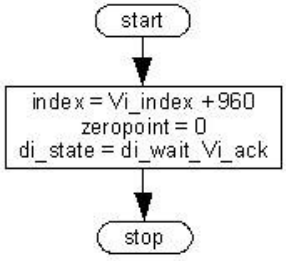


***Figure 45*** *Flowchart of di_rst*

2. `di_wait_Vi_ack` (Figure 46) which waits for the rising edge of the variable `Vi_ack` to signify that the Nik controller unit has a `Vi_val` value (one of the 64 matrixed values calculated from 32 requantised sub-band samples) for the Di controller. This procedure initialises the `index` variable to the index of the array `Vi_stor` of where to load the first of the 64 matrixed values before moving to the `di_shift` procedure;

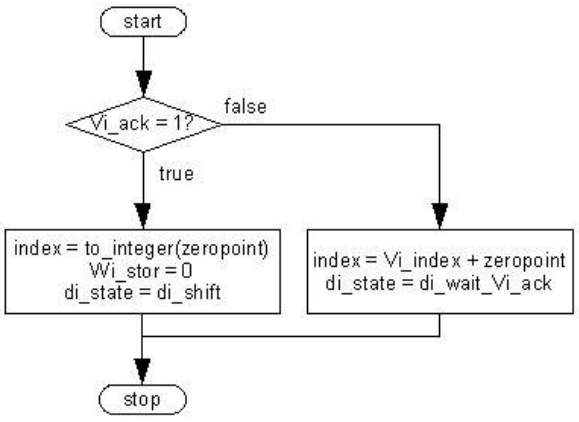


***Figure 46*** *Flowchart of di_wait_Vi_ack*

3. `di_shift` (Figure 47) where the 64 `Vi_val` variables are assigned to the array `Vi_stor` before moving to the `pre_di_mult` procedure;

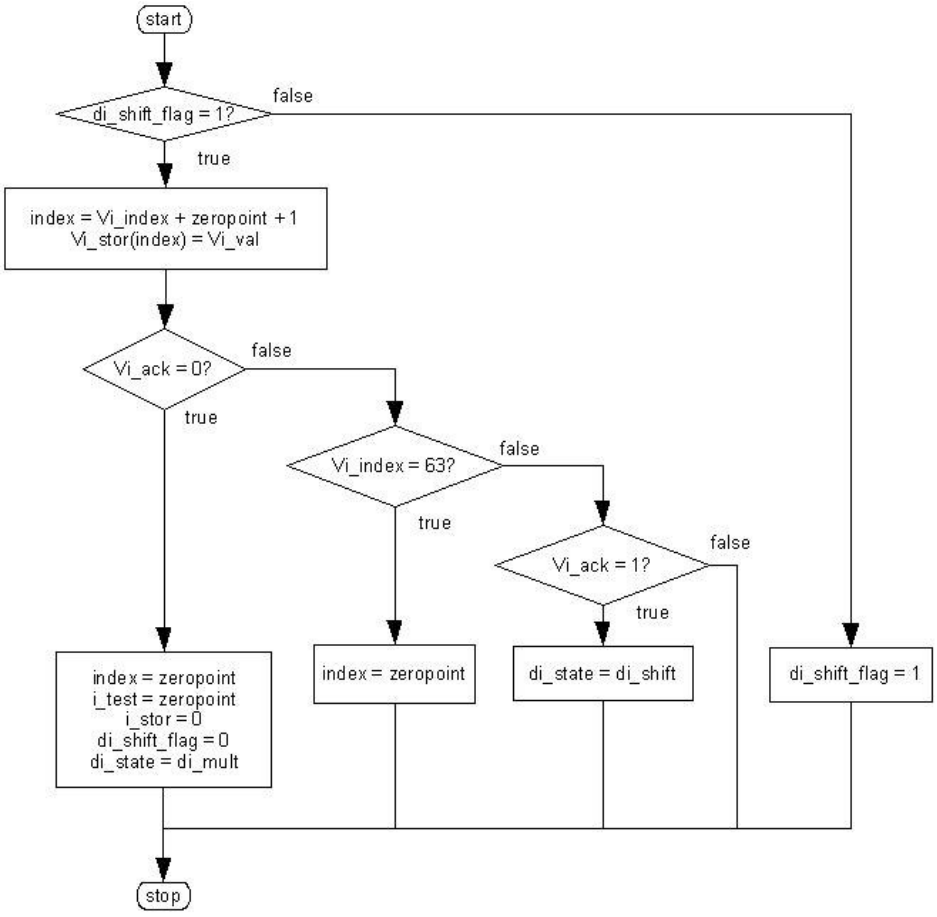


***Figure 47*** *Flowchart of di_shift*

4. `pre_di_mult` reads the sum of previous multiply and accumulates of the requantisied sub-band sample by Nik filter coefficients from the array `Vi_stor` before advancing to the procedure `di_mult`;

5. `di_mult` (Figure 48) performs a multiply and accumlate whereby the requantisied sub-band sample is multiplied by the Di filter coefficient supplied from the Di look up table and added to the sum of previous multiply of the requantisied sub-band sample by Di filter coefficients;

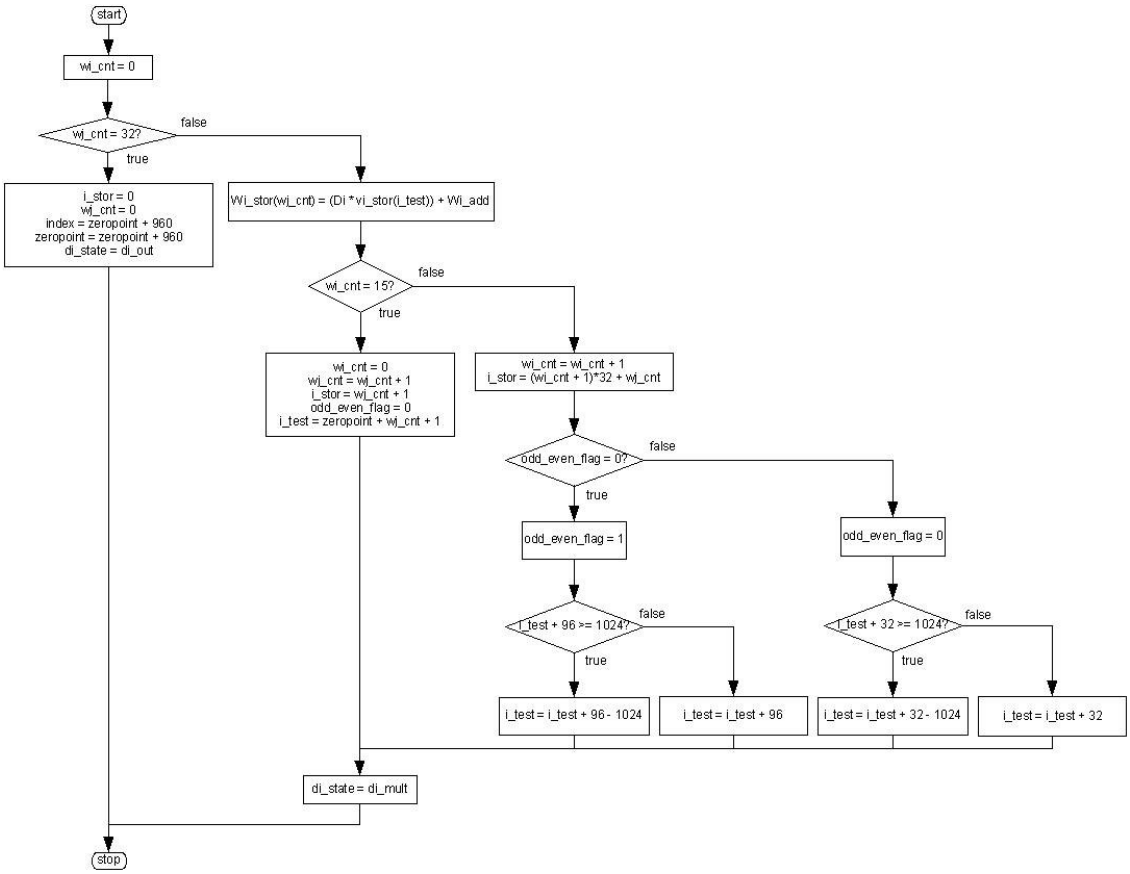


***Figure 48*** *Flowchart of di_mult*

6. `di_out` (Figure 49) which outputs all 32 newly decoded and decompressed PCM samples and returns to the `di_wait_Vi_ack` procedure to wait for the next 64 matrixed values calculated from 32 requantised sub-band samples;

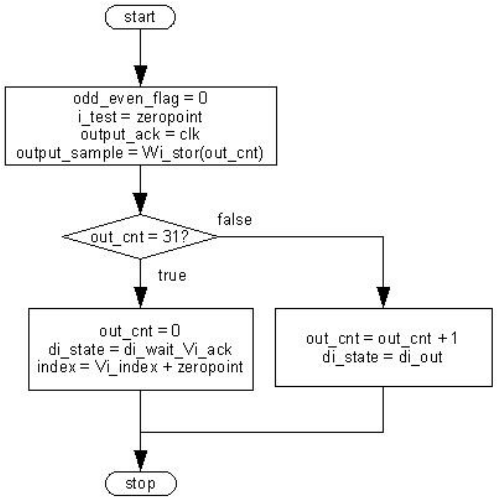


***Figure 49*** *Flowchart of di_out*

### 3.12.4 The Di look up table

The Di look up table is made up of two files of VHDL code as follwos:

- Di look up table shown in **Appendix 1-19** and called `Di_lut.vhd`;
- Di definition file shown in **Appendix 1-20** and called `Di_rom_def.vhd`;

The Di look up table receives an index value from the Di controller indicating which Di filter coefficient is required for the next multiply operation. The Di look up table indexes the Di definition file and returns the required Di filter coefficient. The Di definition file stores the 512 Di filter coefficient values that are supplied with the standard [1]. These Di filter coefficients were then converted to a signed decimal binary representation.

## 3.13 Summary

The VHDL model described in this chapter was developed, at least in part, as a test-bed to analyse the impact of varying the precision of its mathematical operations. This was achieved by allowing the number of bits used for the fractional binary format used for the storage of variables to be increased or decreased through the use of a spreadsheet file and a control file. The spreadsheet file was designed so that the number of bits required for each of the multiplying variables could be entered and the corresponding number of bits for the other variables would be calculated. This spreadsheet was designed in such a way that after the number of bits for the other variables had been calculated the cells of the spreadsheet could be copied and pasted into a VHDL file. The other files in the VHDL model then referenced this file for the number of bits for the other variables.

It was also an objective during the development of the VHDL model to explore methods of either reducing the required hardware resources or minimzing the time required to decode a compressed sample. For example, in the Polyphase Sythensis Sub-band Filter described here, the number of arrays of memory were dramatically decreased compared with the number in the original reference design. A filter implemented according to the standard, requires the resources of a pair of 32 element arrays, a single 64 element array, a pair of 512 element arrays and a single 1024 element array. In comparison the VHDL model uses an implementation of the Polyphase Synthesis Sub-band Filter that uses only a pair of 32 element arrays, a single 64 element array and a single 1024 element array.

# Chapter 4. Performance Evaluation

## 4.1 Introduction

In this chapter, an analysis of the VHDL model of the MPEG 1 Layer I audio decoder is presented. The VHDL model that was described in the previous chapter was first compared to a reference C model [3] supplied with the 11172-3 standard [1] to determine if it meets the specifications set down in the 11172-4 standard [2]. To achieve this, the C model was modified to write test data to a text file at the various points during the decoding process. At corresponding points in the VHDL model, data was also written to text files and the differences between these text files were examined using a standard spreadsheet program (an example is shown in **Appendix 6**). The bit lengths at chosen points in the hardware decoder were then altered and the results compared to the C model, to guage both the overall numerical accuracy as well as the accuracy of results at specific points in the process. Finally, the VHDL model was synthesised for implementation into a FPGA using the area and power evaluation in the Xilinx WebPack software in order to find out the number of flip-flops used for specific bit lengths. This value can then be interpreted as an alias for relative size in the decoder system.

## 4.2 Calculation Methodology

The compliance testing for this thesis was undertaken in two parts. Firstly a compressed file of MPEG 1 Layer I data (`FL4.mp3` supplied with the standard [2]) was decoded by both the reference C model and the VHDL model under test. The

output from both the C and VHDL models were stored and the absolute and RMS errors calculated as described below. The compressed audio waveform applied to both models is derived from a sine wave with a frequency that increases slowly from 20Hz to 10kHz with a maximum amplitude set at 20dB below full scale. The absolute differences between the two models were determined simply by applying the test file (`FL4.mp3` supplied with the standard [2]) to both the VHDL and C models and subtracting values derived at corresponding points in each. The RMS error is givern by

(for n = 1 to N)

$$rms = \sqrt{\frac{1}{N\sum(diff)^2}} \tag{23}$$

where *diff* represents the absolute difference between individual values. The absolute difference (*diff*) and the RMS error (*rms*) were then be used to determine accuracy and compliance, as follows.

**Limited Accuracy ISO/IEC 11172-3 audio decoder**

The only condition that has to be met for an audio decoder to be considered a *limited accuracy* ISO/IEC 11172-3 audio decoder according to the standard [2], is that the RMS error is less than $\frac{1}{(2^{11}\times 12^{0.5})} = 1.409547\times 10^{-4}$. Once the RMS level of the output error is obtained it can be simply compared to the constant $1.409547\times 10^{-4}$. If the RMS level of error VHDL model is less than the constant it can be said to be a limited accuracy ISO/IEC 11172-3 audio decoder.

**Fully Compliant ISO/IEC 11172-3 audio decoder**

On the other hand, for an audio decoder model to be fully compliant with ISO/IEC 11172-3 [2], it must meet two conditions. Firstly, the maximum absolute error has to be less than $\frac{1}{2^{14}} = 6.10351 \times 10^{-5}$, while the second condition requires the RMS error to be less than $\frac{1}{(2^{15} \times 12^{0.5})} = 8.809666 \times 10^{-6}$.

**Measurement of Errors vs Resolution**

The objective of this part of the tesing was to explore the relationship between the resolution and the error performance and to find the specific resolution points where the VHDL model just meets either limited or full compliance. The first stage of these tests focussed only on the reference C model provided in the standard [3]. This was modified to vary the number of bits used to represent the various multipliers forming the decoder. Each multiplication block within the reference C model is expressed in a floating point format. The number of bits used to represent each of these blocks was then modelled in a spreadsheet file and finally their operands were converted into signed decimal binary format. To check the accuracy of the conversion from floating-point to the signed decimal binary format, the signed decimal binary format values were then converted back into a floating-point format. By limiting the number of bits used to convert the signed decimal binary format values back into a floating-point format, each operand could be modelled at a particular bit length in a fairly straightforward manner. Following this step the fixed resolution signed decimal binary format was transferred into the VHDL model and its impact re-simulated.

**Synthesis of the VHDL model**

Finally, in order to determine the resources required for the implementation of the VHDL model, the design was synthesised using the Xilinx WebPack EDA software. Prior to this, all non-synthesisable code (particular, the file handling statements) had to be removed. The synthesis process determines the resources needed to implement the design using a FPGA, including a count of the flip-flops that are used. This has been used as a relative area metric—to guage the comparative implementation size at various resolutions.

## 4.3 Varying the Model Resolution

As outlined above, this part of the testing explored the relationship between resolution and error performance as well as locating the specific points where the VHDL model just passed either limited or full compliance.

### 4.3.1 Varying nb_mult

As discussed in Chpater 3, `nb` represents the number of bits read for each compressed sub-band sample. Each sample is then rescaled using the variables `nb_mult` and `scalefactor`. At this point, the resolution of `nb_mult` was varied then the test file decoded and decompressed by both the modified C and VHDL models. The absolute and RMS differences between the reference and modified C models as well as between the reference C model and the VHDL model were recorded and are shown in Table 13. The rightmost three columns in the table identify whether the VHDL model achieves full or partial compliance along with the number of flip-flops generated for the FPGA implementation. When the precision of multiplers is between 10 and 13 bits, the

modified C model and the VHDL model can be considered to be limited accuracy ISO/IEC 11172 audio decoders. Similarly, when the precision is increased to 14 bits or above the models become fully compliant.

| Number of bits | Modified C model | | | | VHDL model | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Absolute Error | RMS Error | Full compliance | Limited Accuracy | Absolute Error | RMS Error | Full compliance | Limited Accuracy | Flip-Flops |
| 2 | 1.542E-002 | 3.559E-003 | FALSE | FALSE | 1.542E-002 | 3.559E-003 | FALSE | FALSE | 22169 |
| 6 | 3.701E-003 | 1.541E-003 | FALSE | FALSE | 3.701E-003 | 1.541E-003 | FALSE | FALSE | 22565 |
| 8 | 9.032E-004 | 1.914E-004 | FALSE | FALSE | 9.032E-004 | 1.914E-004 | FALSE | FALSE | 22763 |
| 9 | 4.556E-004 | 1.722E-004 | FALSE | FALSE | 4.556E-004 | 1.722E-004 | FALSE | FALSE | 22862 |
| 10 | 2.340E-004 | 5.708E-005 | FALSE | TRUE | 2.340E-004 | 5.708E-005 | FALSE | TRUE | 22961 |
| 12 | 3.030E-005 | 1.197E-005 | FALSE | TRUE | 3.030E-005 | 1.197E-005 | FALSE | TRUE | 23159 |
| 13 | 3.030E-005 | 1.197E-005 | FALSE | TRUE | 3.030E-005 | 1.197E-005 | FALSE | TRUE | 23258 |
| 14 | 6.670E-006 | 1.773E-006 | TRUE | TRUE | 6.670E-006 | 1.773E-006 | TRUE | TRUE | 23357 |
| 18 | 5.000E-007 | 1.344E-007 | TRUE | TRUE | 5.000E-007 | 1.343E-007 | TRUE | TRUE | 23753 |
| 22 | 2.000E-007 | 2.515E-008 | TRUE | TRUE | 2.000E-007 | 2.515E-008 | TRUE | TRUE | 24149 |
| 26 | 2.000E-007 | 1.133E-008 | TRUE | TRUE | 2.000E-007 | 1.127E-008 | TRUE | TRUE | 24545 |
| 30 | 1.200E-007 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 1.187E-009 | TRUE | TRUE | 24941 |
| 34 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 25337 |
| 38 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 25733 |
| 42 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26129 |
| 46 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26525 |
| 50 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26921 |

***Table 13*** *nb_mult length predictions*

The absolute differences versus the number of bits between the reference and modified C models as well as between the reference C and VHDL models is shown in Figure 50.

From the data in Table 13, it is clear that the two curves, shown in Figure 50, are identical and at a resolution greater than about 30 bits the difference between reference and modified C code models falls to zero. When `nb_mult` reaches 12 or greater, the absolute difference becomes less than $6.10351 \times 10^{-5}$, meeting the second condition for a fully compliant ISO/IEC 11172-3 audio decoder.

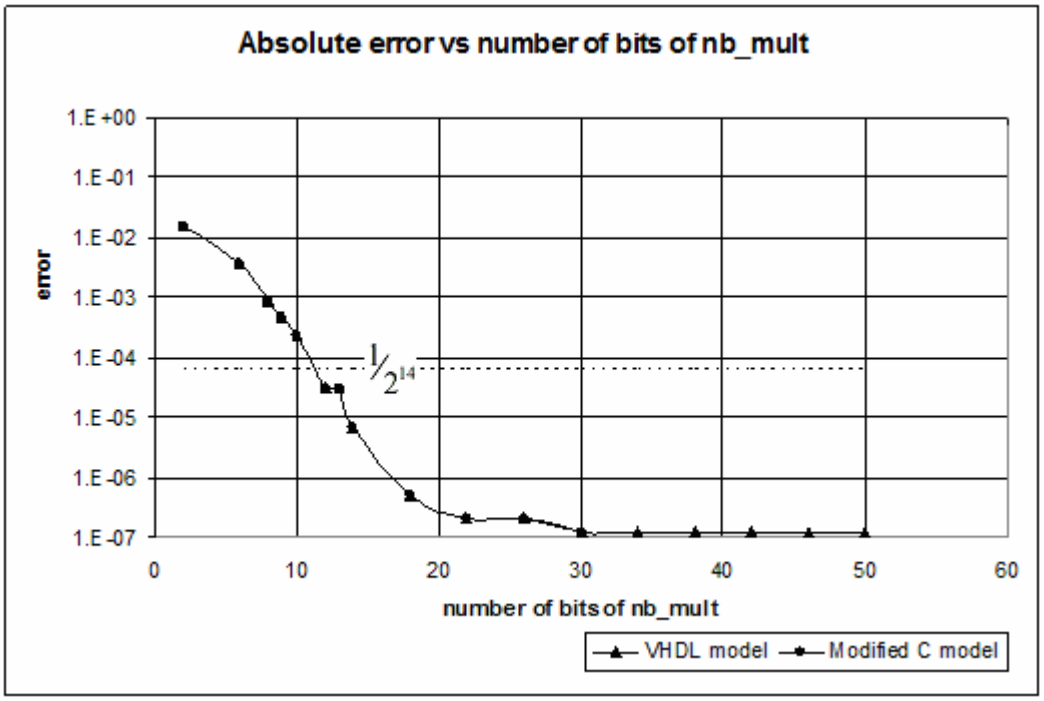


***Figure 50*** *Absolute error versus the number of bits of nb_mult*

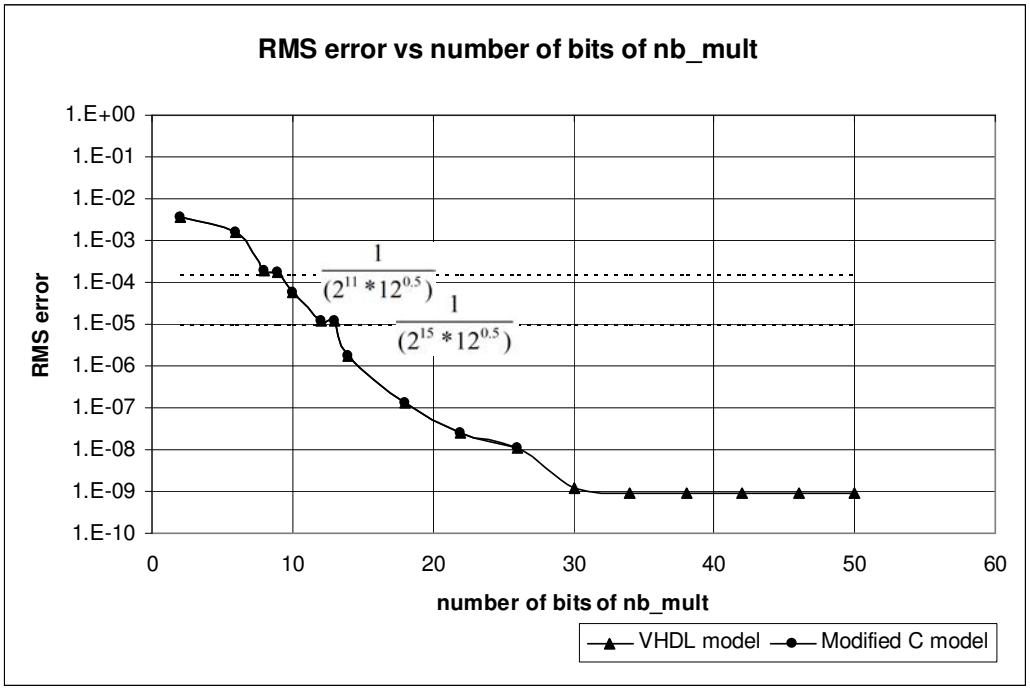


***Figure 51*** *RMS error versus number of bits of nb_mult*

The RMS error is plotted in Figure 51. In this case the RMS level of error falls below 1.409547 x $10^{-4}$ for `nb_mult` less than or equal to 10 bits, indicating that the condition for a limited accuracy ISO/IEC 11172-3 audio decoder has been met. Further, the system becomes fully compliant at 14 bits as the RMS error is less than 8.809666 x $10^{-6}$, thus meeting the second condition.

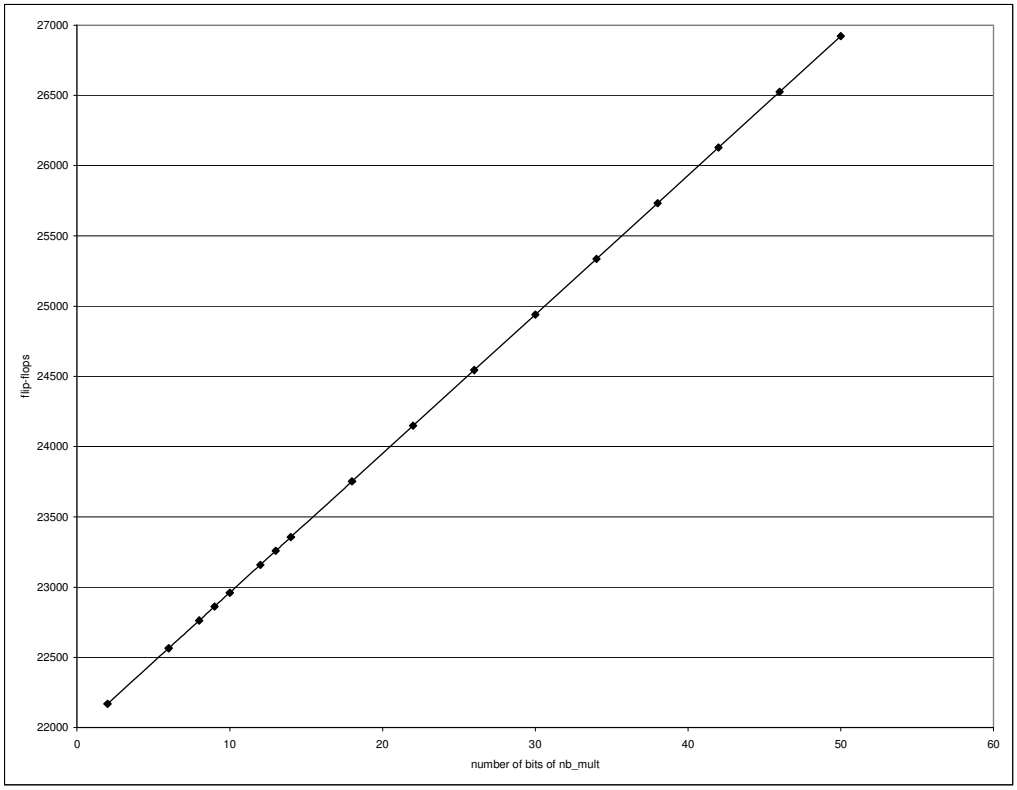


***Figure 52*** *Number of flip-flops versus number of bits of Nb_mult*

Figure 52 shows number of flip-flops required for FPGA implementation across the range of resolutions between 2 and 50 bits. As expected, the graph shows a completely linear relationship between resolution and area.

### 4.3.2 Varying scale factor

In these tests the resolution of the scale factor applied to each compressed sub-band sample was varied. The test file was decoded and decompressed in the same way as before and the absolute and RMS errors were recorded (Table 14). Again, the three rightmost columns identify whether full or partial compliance was achieved for a particular number of bits and the required number of flip-flops for FPGA implementation.

| Number of bits | Modified C model | | | | VHDL model | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Absolute Error | RMS Error | Full compliance | Limited Accuracy | Absolute Error | RMS Error | Full compliance | Limited Accuracy | Flip-Flops |
| 3 | 1.194E-001 | 6.975E-002 | FALSE | FALSE | 1.194E-001 | 6.975E-002 | FALSE | FALSE | 20189 |
| 7 | 8.210E-002 | 1.900E-002 | FALSE | FALSE | 8.210E-002 | 1.900E-002 | FALSE | FALSE | 20585 |
| 11 | 4.073E-003 | 8.342E-004 | FALSE | FALSE | 4.073E-003 | 8.342E-004 | FALSE | FALSE | 20981 |
| 13 | 1.125E-003 | 3.001E-004 | FALSE | FALSE | 1.125E-003 | 3.001E-004 | FALSE | FALSE | 21179 |
| 14 | 5.236E-004 | 9.380E-005 | FALSE | TRUE | 5.236E-004 | 9.380E-005 | FALSE | TRUE | 21278 |
| 15 | 2.904E-004 | 5.408E-005 | FALSE | TRUE | 2.904E-004 | 5.408E-005 | FALSE | TRUE | 21377 |
| 17 | 7.045E-005 | 1.598E-005 | FALSE | TRUE | 7.045E-005 | 1.598E-005 | FALSE | TRUE | 21575 |
| 18 | 4.370E-005 | 1.382E-005 | FALSE | TRUE | 4.370E-005 | 1.382E-005 | FALSE | TRUE | 21674 |
| 19 | 2.313E-005 | 7.218E-006 | TRUE | TRUE | 2.313E-005 | 7.218E-006 | TRUE | TRUE | 21773 |
| 23 | 2.030E-006 | 4.390E-007 | TRUE | TRUE | 2.030E-006 | 4.390E-007 | TRUE | TRUE | 22169 |
| 27 | 2.000E-007 | 4.242E-008 | TRUE | TRUE | 2.000E-007 | 4.242E-008 | TRUE | TRUE | 22565 |
| 31 | 2.000E-007 | 1.056E-008 | TRUE | TRUE | 2.000E-007 | 1.051E-008 | TRUE | TRUE | 22961 |
| 35 | 1.200E-007 | 1.953E-009 | TRUE | TRUE | 1.200E-007 | 2.422E-009 | TRUE | TRUE | 23357 |
| 39 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 23753 |
| 43 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 24149 |
| 47 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 24545 |
| 51 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 24941 |
| 55 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 25337 |
| 59 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 25733 |
| 63 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26129 |
| 67 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26525 |
| 71 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26921 |

***Table 14*** *Scale factor length predictions*

These results are plotted in Figure 53 and Figure 54. Between about 7 and 27 bits, there is a virtually linear relationship between RMS error and resolution. At 18 bits or

larger, the absolute difference between the reference model and both the modified C and VHDL models is less than 6.10351 x $10^{-5}$, so that the second compliance condition has been met. It can also be seen that when the number of bits of the scale factor multiplier is around 39 bits or larger, there is essentially no difference between the output of the reference and modified C models—the RMS error falls to almost zero. In this case, it can be seen that between 14 and 18 bits the modified C model and VHDL model can be considered to represent a limited accuracy ISO/IEC 11172 audio decoder, whereas at or above 19 bits the modified C model is fully compliant. As before Figure 55 shows the a linear increase of the number of flip-flops required for FPGA implementation as the number of bits is increased.

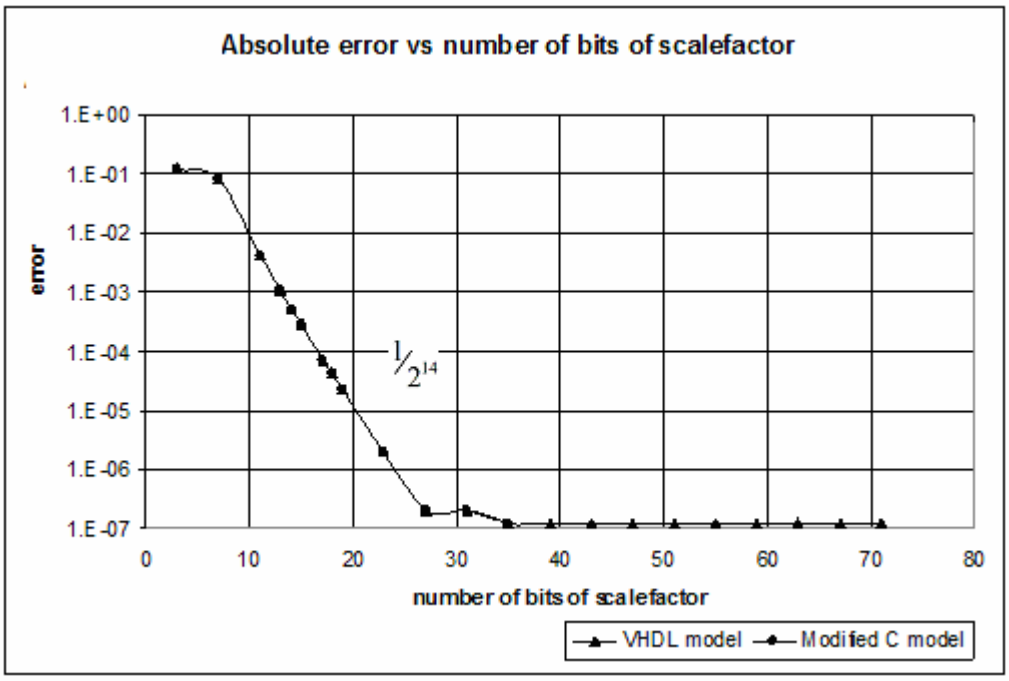


***Figure 53*** *Absolute error versus the number of bits of scalefactor*

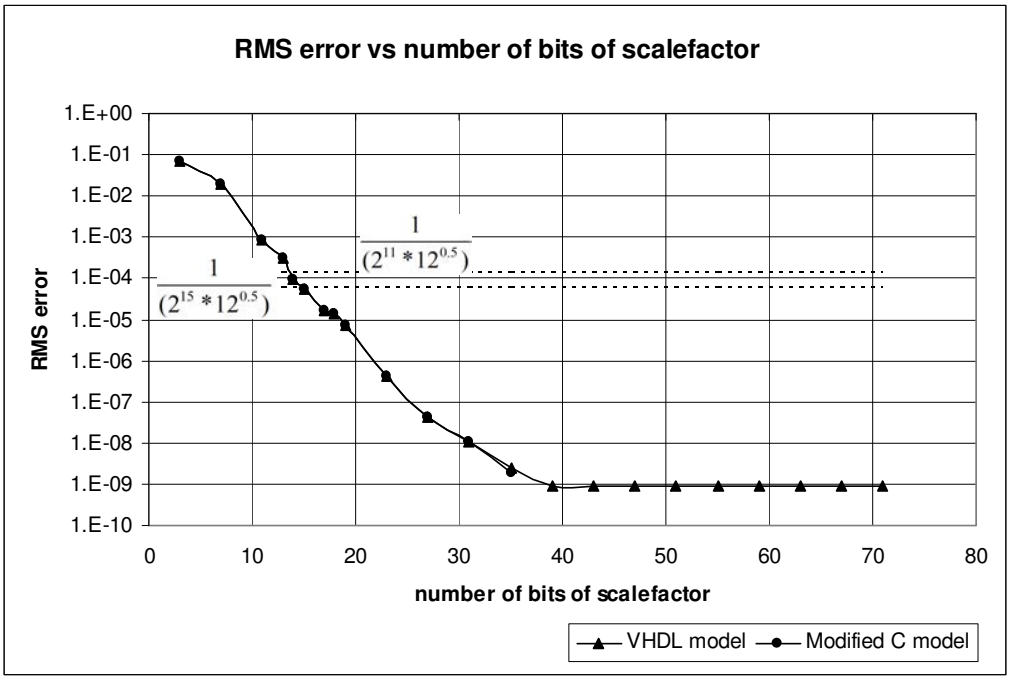


***Figure 54*** *RMS error versus number of bits of scale factor*

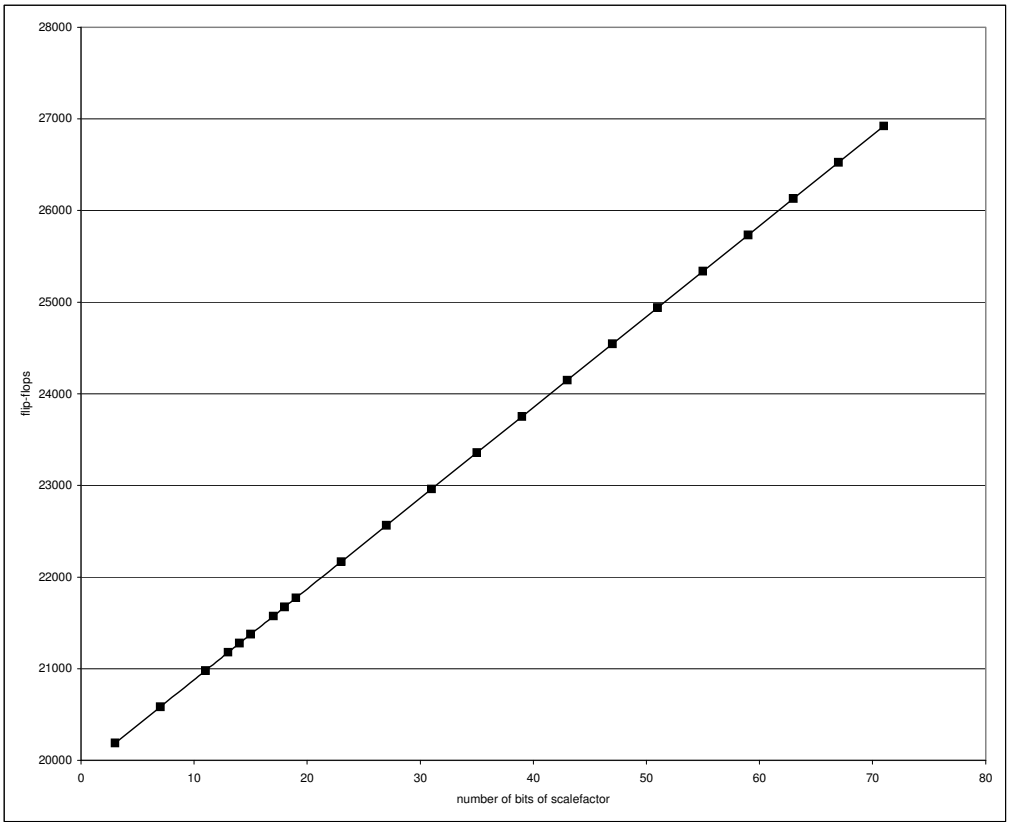


***Figure 55*** *Number of flip-flops versus number of bits of scalefactor*

### 4.3.3 Varying Nik filter coefficient

At this point, the resolution of the Nik filter coefficient variable was varied and the maximum and RMS error measured. The results are shown in Table 15 along with the compliance level achieved.

| Number of bits | Modified C model | | | | VHDL model | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Absolute Error | RMS Error | Full compliance | Limited Accuracy | Absolute Error | RMS Error | Full compliance | Limited Accuracy | Flip-Flops |
| 1 | 1.194E-001 | 6.975E-002 | FALSE | FALSE | 3.098E-001 | 1.093E-001 | FALSE | FALSE | 20981 |
| 5 | 2.126E-002 | 7.688E-003 | FALSE | FALSE | 2.481E-002 | 7.305E-003 | FALSE | FALSE | 21377 |
| 9 | 1.257E-003 | 4.815E-004 | FALSE | FALSE | 1.551E-003 | 4.535E-004 | FALSE | FALSE | 21773 |
| 10 | 6.036E-004 | 2.302E-004 | FALSE | FALSE | 7.752E-004 | 2.279E-004 | FALSE | FALSE | 21872 |
| 11 | 3.095E-004 | 1.127E-004 | FALSE | TRUE | 3.877E-004 | 1.156E-004 | FALSE | TRUE | 21971 |
| 13 | 8.140E-005 | 2.964E-005 | FALSE | TRUE | 9.691E-005 | 2.793E-005 | FALSE | TRUE | 22169 |
| 14 | 4.210E-005 | 1.399E-005 | FALSE | TRUE | 4.840E-005 | 1.436E-005 | FALSE | TRUE | 22268 |
| 15 | 2.000E-005 | 7.300E-006 | TRUE | TRUE | 2.420E-005 | 7.105E-006 | TRUE | TRUE | 22367 |
| 17 | 5.700E-006 | 1.852E-006 | TRUE | TRUE | 6.080E-006 | 1.823E-006 | TRUE | TRUE | 22565 |
| 21 | 4.000E-007 | 1.261E-007 | TRUE | TRUE | 4.700E-007 | 1.215E-007 | TRUE | TRUE | 22961 |
| 25 | 2.000E-007 | 2.634E-008 | TRUE | TRUE | 2.000E-007 | 2.634E-008 | TRUE | TRUE | 23357 |
| 29 | 1.200E-007 | 7.390E-009 | TRUE | TRUE | 1.200E-007 | 6.878E-009 | TRUE | TRUE | 23753 |
| 33 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 24149 |
| 37 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 24545 |
| 41 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 24941 |
| 45 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 25337 |
| 49 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 25733 |
| 53 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26129 |
| 57 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26525 |
| 61 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26921 |

***Table 15*** *Nik filter coefficient length predictions*

When the precision of the Nik filter coefficient is between 11 and 14 bits in length (as seen in Table 15), the modified C model and the VHDL model can be considered to be a limited accuracy ISO/IEC 11172 audio decoder. At 15 bits or above the models become fully compliant. The RMS error becomes insignificant when the Nik resolution reaches 33 bits.

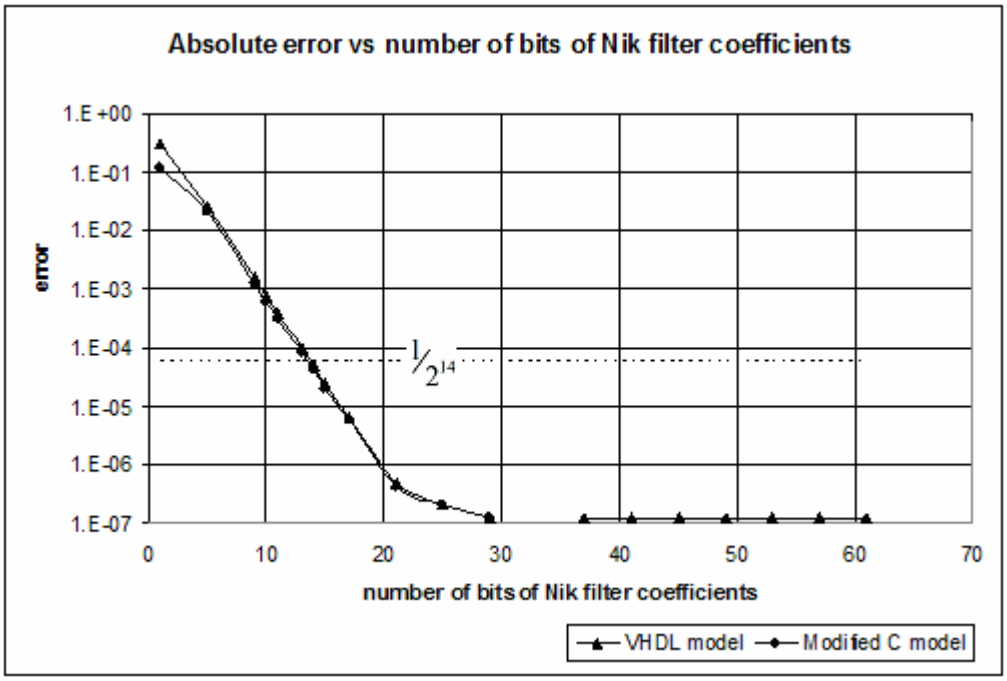


***Figure 56*** *Absolute error versus the number of bits of Nik filter coefficients*

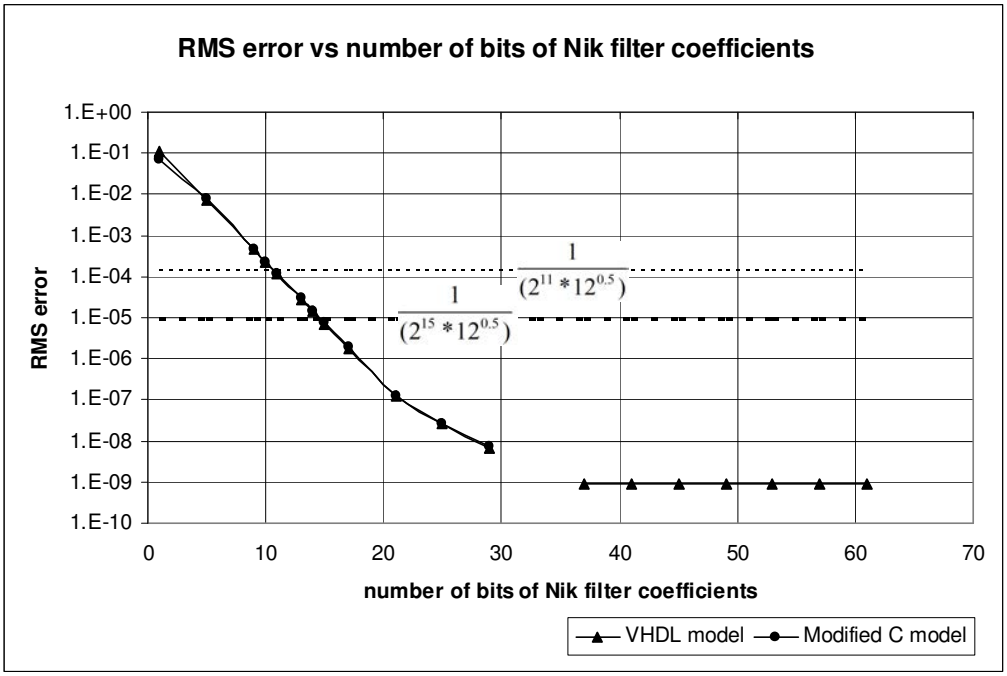


***Figure 57*** *RMS error versus number of bits of Nik filter coefficients*

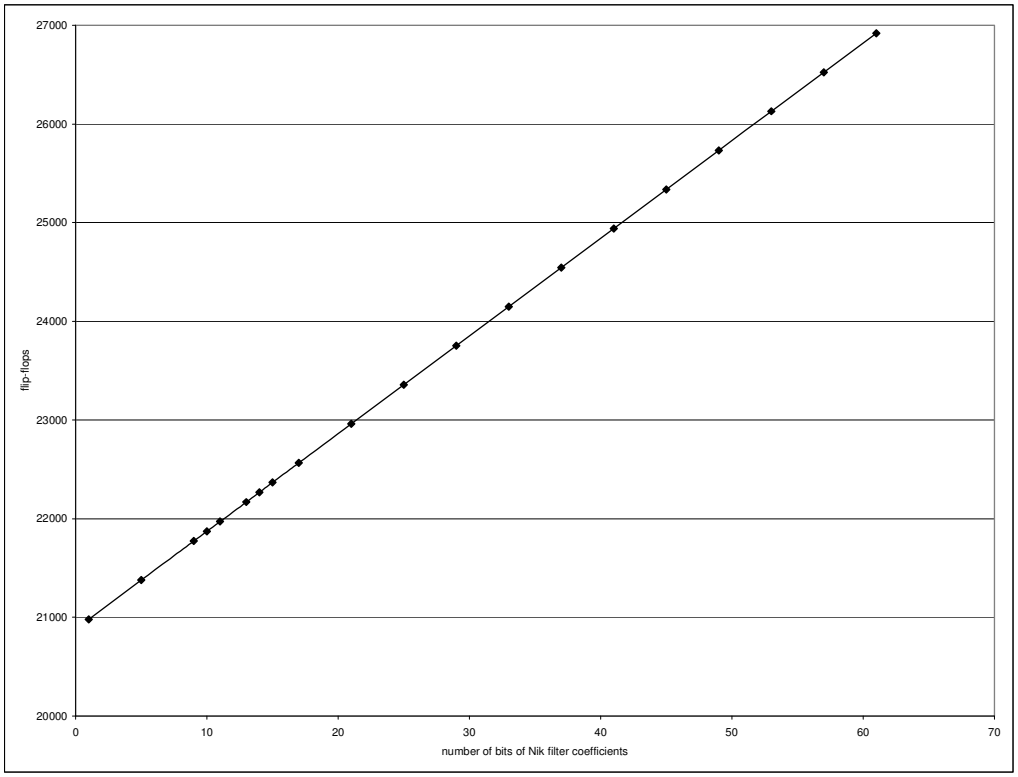


***Figure 58*** *Number of flip-flops used versus number of bits of Nik coefficient*

Once again Figure 58 shows the a linear increase of the number of flip-flops required for FPGA implementation when the number of bits is increased.

### 4.3.4 Varying Di filter coefficient

The resolution of the Di filter coefficient variable is varied in the same manner and the various errors and compliances recorded (Table 16). It can be seen that when the Di filter coefficient is between 13 and 15 bits the models can be considered a limited accuracy ISO/IEC 11172 audio decoders and although the second compliance condition is met at 15 bits or larger they become fully compliant only at 16 bits or greater. At 34 bits or larger, the absolute error between the output of the reference and modified C models reduces to zero.

| Number of bits | Modified C model | | | | VHDL model | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Absolute Error | RMS Error | Full compliance | Limited Accuracy | Absolute Error | RMS Error | Full compliance | Limited Accuracy | Flip-Flops |
| 2 | 1.194E-001 | 4.727E-002 | FALSE | FALSE | 4.675E-001 | 1.299E-001 | FALSE | FALSE | 25073 |
| 6 | 1.765E-002 | 4.769E-003 | FALSE | FALSE | 2.692E-002 | 7.542E-003 | FALSE | FALSE | 25205 |
| 10 | 1.452E-003 | 4.237E-004 | FALSE | FALSE | 1.818E-003 | 4.613E-004 | FALSE | FALSE | 25337 |
| 11 | 8.205E-004 | 2.051E-004 | FALSE | FALSE | 8.068E-004 | 2.270E-004 | FALSE | FALSE | 25370 |
| 12 | 8.205E-004 | 2.051E-004 | FALSE | FALSE | 3.886E-004 | 1.127E-004 | FALSE | TRUE | 25403 |
| 13 | 2.067E-004 | 5.504E-005 | FALSE | TRUE | 2.146E-004 | 5.630E-005 | FALSE | TRUE | 25436 |
| 14 | 1.067E-004 | 2.808E-005 | FALSE | TRUE | 9.525E-005 | 2.809E-005 | FALSE | TRUE | 25469 |
| 15 | 5.567E-005 | 1.433E-005 | FALSE | TRUE | 5.377E-005 | 1.411E-005 | FALSE | TRUE | 25502 |
| 16 | 2.659E-005 | 7.122E-006 | TRUE | TRUE | 2.623E-005 | 6.949E-006 | TRUE | TRUE | 25535 |
| 18 | 7.990E-006 | 2.116E-006 | TRUE | TRUE | 7.990E-006 | 2.116E-006 | TRUE | TRUE | 25601 |
| 22 | 5.000E-007 | 1.403E-007 | TRUE | TRUE | 5.000E-007 | 1.418E-007 | TRUE | TRUE | 25733 |
| 26 | 2.000E-007 | 2.796E-008 | TRUE | TRUE | 2.000E-007 | 2.874E-008 | TRUE | TRUE | 25865 |
| 30 | 2.000E-007 | 6.777E-009 | TRUE | TRUE | 2.000E-007 | 6.893E-009 | TRUE | TRUE | 25997 |
| 34 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 1.237E-009 | TRUE | TRUE | 26129 |
| 38 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26261 |
| 42 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26393 |
| 46 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26525 |
| 50 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26657 |
| 54 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26789 |
| 58 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 26921 |
| 62 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 27053 |
| 66 | 0.000E+000 | 0.000E+000 | TRUE | TRUE | 1.200E-007 | 8.748E-010 | TRUE | TRUE | 27185 |

***Table 16*** *Di filter coefficient length predictions*

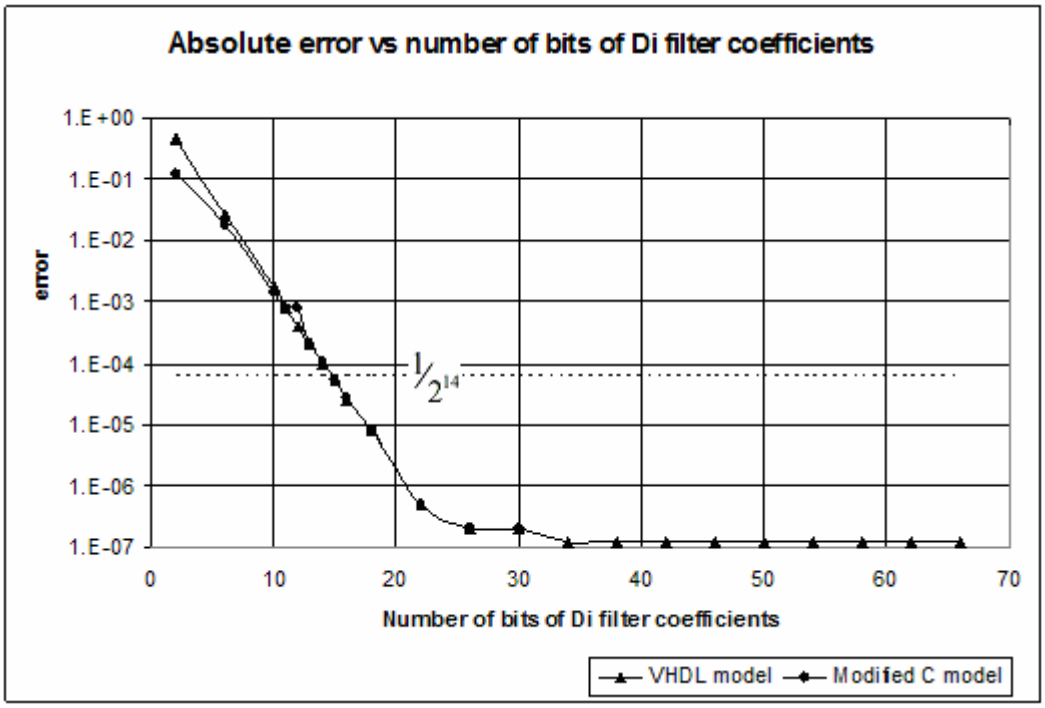


***Figure 59*** *Absolute error versus the number of bits of Di filter coefficients*

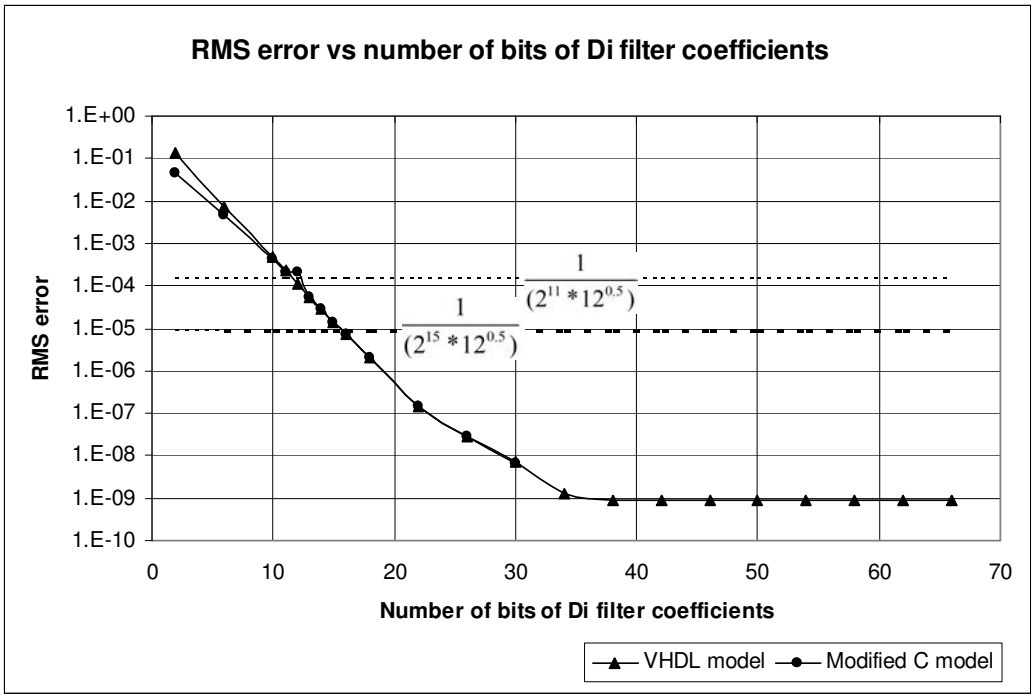


***Figure 60*** *RMS error versus number of bits of Di filter coefficient*

Once again, Figure 61 shows a linear relationship between resolution and area.

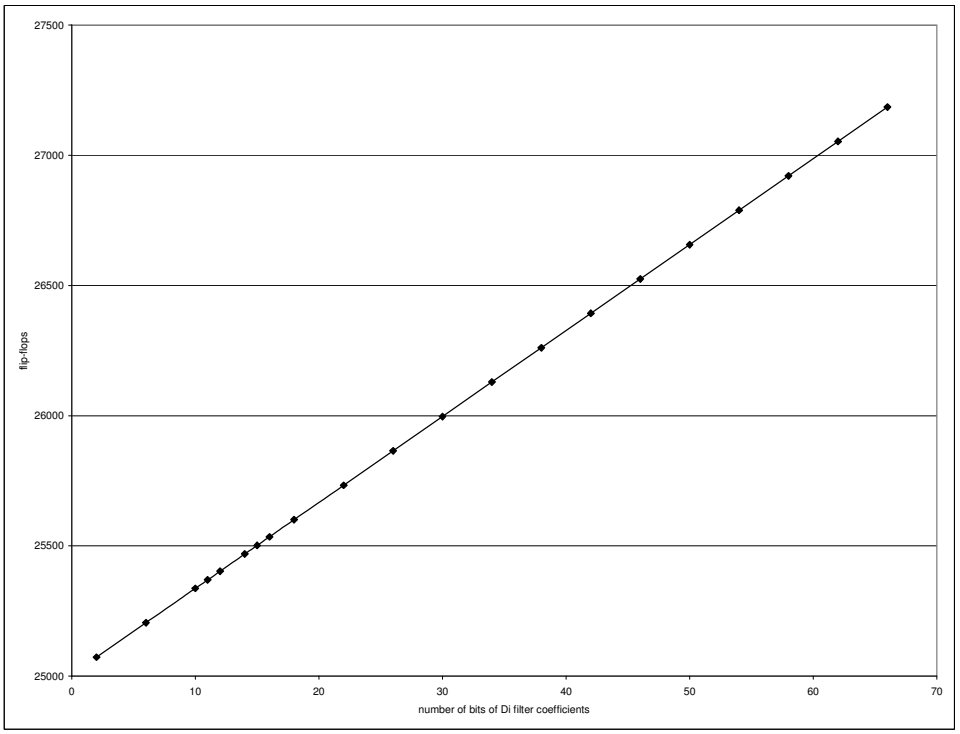


***Figure 61*** *Number of flip-flops versus number of bits of Di filter coefficient*

## 4.4 Compliance vs. Resolution Results

In the previous sections, the bit lengths of the four multipliers were varied independently and the absolute and RMS errors between the reference model and both the modified C and VHDL models were measured in order to find the bit lengths required to meet both full and limited accuracy. This is summarized in Table 17.

| Variable | Full Representation | Modified C Model | | VHDL Model | |
|---|---|---|---|---|---|
| | | Full compliance | Limited Accuracy | Full compliance | Limited Accuracy |
| nb_mult | 53 | 14 (26.4%) | 10 (18.9%) | 14 (26.4%) | 10 (18.9%) |
| scalefactor | 74 | 19 (25.7%) | 14 (18.9%) | 19 (25.7%) | 14 (18.9%) |
| Nik | 65 | 15 (23.1%) | 11 (16.9%) | 15 (23.1%) | 11 (16.9%) |
| Di | 70 | 16 (22.9%) | 13 (18.6%) | 16 (22.9%) | 12 (17.1%) |

***Table 17*** *Summary of minimum bit lengths required from modified C model*

When the predicted results from the modified C model are compared to the results from the VHDL model it can be seen that there is only one difference. The number of bits required for the representation of the Di filter coefficient model to meet the limited compliance of a MPEG 1 Layer audio decoder is predicted to require a length of 13 bits compared to the 12 bits of the VHDL model. This is most likely a rounding issue between the two models.

It can be seen from Table 17 that each of the multiply units can have their bit lengths reduced by about 75% from the reference case and still achieve full compliance. A reduction in bit length of 82% is still sufficient to meet limited compliance.

As a final test to determine if the resolutions of the various multiply units in the VHDL model operate independently, all the variables were initialised to their limited accuracy thresholds shown in Table 17. Each of the four variables was then decremented by one

in turn and the compliance terms remeasured. As shown in the first four rows of Table 18, the model no longer meets even the limited accuracy criterion under any of these conditions. In the same way, all four variables were set to their minimum values for full compliance and then each is independently decremented. In this case (see the lower four rows of Table 18), the VHDL model loses full compliance but still meets the limited accuracy requirements. From this, it appears that the variables operate independently from each other and that the audio decoder will be compliant as long as the bit lengths are set to the minimum values given in Table 17.

| Variable | Number of Bits | VHDL model | | | |
|---|---|---|---|---|---|
| | | Absolute Error | RMS Error | Full Compliance | Limited Accuracy |
| Nb | 9 | 8.674E-004 | 1.999E-004 | FALSE | FALSE |
| Scalefactor | 13 | 6.622E-004 | 1.858E-004 | FALSE | FALSE |
| Nik | 10 | 7.914E-004 | 1.793E-004 | FALSE | FALSE |
| Di | 11 | 7.899E-004 | 1.789E-004 | FALSE | FALSE |
| Nb | 15 | 4.912E-005 | 1.304E-005 | FALSE | TRUE |
| Scalefactor | 18 | 4.054E-005 | 1.114E-005 | FALSE | TRUE |
| Nik | 14 | 4.530E-005 | 1.152E-005 | FALSE | TRUE |
| Di | 15 | 4.054E-005 | 1.114E-005 | FALSE | TRUE |

***Table 18*** *Results of independently varying variables*

At the final stage of testing, the Xilinx WebPack software was used to find the resources required for each of the bit lengths. Table 19 shows the number of flip-flops synthesised as a result of setting the bit lengths to their values in Table 17. From both of these tables it is obvious that greater reductions can be achieved earlier in the decoding process, i.e., it becomes progressively more difficult to trim the multiplier bit lengths at later stages in the decode pipeline.

| Variable | Full Representation | VHDL Model | |
|---|---|---|---|
| | | Full compliance | Limited Accuracy |
| nb_mult | 27317 | 23357 | 22961 |
| Scalefactor | 27317 | 21773 | 21278 |
| Nik | 27317 | 22367 | 21971 |
| Di | 27317 | 25535 | 25403 |
| Best case - all multipliers set to minimum | 27317 | 11081 | 9662 |

***Table 19*** *Summary of resources used*

## 4.5 Summary

Testing of the VHDL model involved the following steps:

- Testing the VHDL model against the unmodified reference C model to ensure compliance of the model;
- Modifing the bit lengths of the four multiplication stages of MPEG 1 Layer I decoding (`nb`, `scalefactor`, Nik filter coefficient and Di filter coefficient) in the modified C model and calculating the resulting absolute and RMS error terms;
- Modifing the bit lengths of the four multiplication stages of MPEG 1 Layer I decoding in the VHDL model and calculating error terms in the same manner;
- Synthesisng the the VHDL model at the different bit legnths of the multipliers to determine the hardware resources used;

When the output from the VHDL model of the MPEG 1 Layer I audio decoder was compared to the output from the C model [3] supplied with the 11172-3 standard [1]. It

was found that it met the specifications set down in 11172-4 standard [2] for compliance testing of a MPEG audio decoder. This was initial verification that the VHDL model was operating correctly.

Following the initial verification, the same spreadsheets that were used to calculate the signed binary form of the multipliers in the design of the VHDL model were used to calculate the values of the multipliers with reduced bit lengths. The bit lengths of the four multipliers were then varied independently and the differences and RMS level of error between the output of the modified C model and the reference C models were calculated to determine the bit lengths required to meet full accuracy and limited accuracy. It was found that each of the multiply units could operate with its resolution reduced by 75% from the reference and still achieve full compliance. Between 14 and 19 bits were required to acheieve full compliance. A reduction in bit length of 82%, 10—13 bits, would still meet limited compliance criterion. The greatest reductions were achieved earlier in the decoding process. It is worth noting that this implies that an implementation using standard 16-bit microprocessor would have little trouble achieving the limited accuracy criterion, but would need to use double-precision multiplication in order to reach full compliance.

# Chapter 5. Summary and Conclusions

## 5.1 Summary

The work in this thesis has focussed on the design and analysis of a MPEG (Moving Picture Experts Group) 1, Layer I audio decompression hardware unit as defined in the standard: ***ISO/IEC 11172-4:1993 Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s - Part 3 Audio*** [1]. As one of the main objectives of this work has been to determine the most efficient bit lengths for the various multiplication stages in the decoder whilst still meeting compliance standards, a flexible VHDL model of the decoder was built and its capabilities for decoding and decompression tested with various bit lengths.

While there is a wealth of information on variable bit length decoding, very little applies directly to audio decompression and the tradeoffs between resource utilisation, resolution, accuracy and performance. Research to date has typically been based on implementation in microprocessors and because of this, existing implementations operate with bit lengths fixed at an integer multiple of the microprocessor word size. Due to its high commercial value research into these areas is not in the open literature.

The behaviour of the VHDL model has been evaluated using a standard audio bit stream and compared with a reference model written in C, both derived from the standard.

This process has involved the following four main steps:

1. Testing the VHDL model against the unmodified reference C model to ensure its basic compliance;

2. Modifing the bit lengths of the four multiplication stages in the modified C model and calculating the absolute and RMS error values;

3. Modifing the bit lengths of the four multiplication stages in the VHDL model and calculating the corresponding error values;

4. Synthesisng the the VHDL model at these different bit lengths to determine the necessary hardware resources.

## 5.2 Conclusions

Apart from its basic usefulness as within potential embedded products, the VHDL model developed for this thesis was intended, at least in part, as a test-bed to analyse the impact of varying the precision of its mathematical operations. This was achieved by allowing the number of bits used for the fractional binary format used for the storage of variables to be increased or decreased. The specific bit lengths were determined by the use of a spreadsheet that generated a control file for the VHDL code. The spreadsheet file was designed so that the number of bits required for each of the multiplication variables could be entered and the corresponding number of bits for the other variables would be calculated. After the number of bits for the other variables had been calculated the cells of the spreadsheet could be copied and pasted into a VHDL file. The other files in the VHDL model then referenced this file to determine the number of bits for the remaining resolution variables.

As indicated above, the VHDL model was analysed by firstly a test waveform from the ISO/IEC standard and comparing the result to the output of a C-language reference model. The models were compared by modifying each to write test data to text files at corresponding points and examining the differences using the spreadsheet program. Once the VHDL model was found to be compliant, the bit lengths of the various multiplication stages were then increased and decreased. The same test waveform was then applied to determine if the model was still compliant at the new bit lengths of the multipliers. This was then repeated until the smallest bit lengths that would still allow compliance.

## 5.3 Future Research

This research has dealt solely with MPEG 1 Layer I audio decoding. It could therefore form the basis of further research into MPEG 1 Layer II and Layer III audio decoding. Both Layer II and Layer III would benefit from a similar type of investigation into bit length versus accuracy and implementation area.

Similar techniques and procedures could also be used to investigate implementations of the various MPEG video compression standards as well as other audio and video compression systems.

# References


[1] ISO/IEC, "Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 3: Audio," *11172*, vol. 3, 1993.

[2] ISO/IEC, "Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 4: Compliance testing," *11172*, vol. 4, 1995.

[3] ISO/IEC, "dist10 Source Code Package," ftp://ftp.tnt.uni-hannover.de/pub/MPEG/audio/mpeg2/software/, 1996.

[4] Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, 1997.

[5] Ronald E. Schafer and L. R. Rabiner, "Digital Representations of Speech Signals," *Proceedings of the IEEE*, vol. 63, pp. 662 - 677, 1975.

[6] N.S. Jayant and P. Noll, *Digital Coding of Waveforms Principles and Applications to Speech and Video*. Englewood Clifs, New Jersey, U.S.A.: Prentice-Hall, 1984.

[7] D. Labratories, "Dolby B, C, AND S Noise Reduction Systems: Making Cassettes Sound Better."

[8] H. Nyquist, "Certain Topics in Telegraph Transmission Theory," presented at Winter Convention of the A. I. E. E, New York, NY, U.S.A., 1928.

[9] A. S. Spanias, "Speech Coding: A Tutorial Review," *Proceedings of the IEEE*, vol. 82, pp. 1541 - 1582, 1994.

[10] P. Elias, "Predictive Coding," *IRE Transactions—Information Theory*, pp. 16 - 33, 1955.

[11] H. Dudley, "The vocoder," *Bell Labs. Rec*, vol. 17, pp. 122, 1939.

[12] B. M. Oliver, J. R. Pierce, and C. E. Shannon, "The Philosophy of PCM," *Proceedings of the I.R.E.*, vol. 36, pp. 1324 - 1331, 1948.

[13] B. S. Atal, "The History of Linear Prediction," in *Signal Processing Magazine*, vol. 23, 2006, pp. 154 - 161.

[14] ISO/IEC, "Information technology - Generic coding of moving pictures and associated audio information - Part 3: Audio," vol. 13818-3, 1998.

[15] ISO/IEC, "Information technology - Coding of audio-visual objects - Part 3: Audio," *14496*, vol. 3, 1999.

[16] Peter Noll, "MPEG Digital Audio Coding," in *IEEE Signal Processing Magazine*, vol. 14, 1997, pp. 59 - 81.

[17] Davis Pan, "A Tutorial on MPEG/Audio compression," in *IEEE Multimedia Magazine*, 1995, pp. 60 - 74.

[18] Seong Hwan Cho, Thucydides Xanthopoulos, and Anantha P. Chandrakasan, "A Low Power Variable Length Decoder for MPEG-2 Based on Nonuniform Fine-Grain Table Partitioning," *IEEE Transactions on Very Large Scale Intergration Systems*, vol. 7, pp. 249 - 257, 1999.

[19] Jari Nikara, Stamatis Vassiliadis, Jarmo Takala, and Petri Liuha, "FPGA-Based Variable Length Decoders," *Proceedings of VLSI-SOC 2003*, pp. 437 - 441, 2003.

[20] Konstantinos Konstantinides, "Fast Subband Filtering in MPEG Audio Coding," *IEEE Signal Processing Letters*, vol. 1, pp. 26 - 28, 1994.

[21] Byeong Gi Lee, "A new algorithm for computing the discrete cosine transform," *IEEE Transactions on Acoustic, Speech and Signal Processing*, vol. 32, pp. 1243 - 1245, 1984.
[22] Wen Hsiung Chen, C. Harrison Smith, and S. C. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform," *IEEE Transactions on Communications*, vol. 25, pp. 1004 - 1009, 1977.
[23] World DAB forum, http://www.worlddab.org/technology_faq.php.
[24] Min-Seop Jeong, Seehyun Kim, Jongseo Sohn, and Wonyony Sung, "Wordlength Optimization of an MPEG-2 Audio Decoder," presented at Proceedings of IEEE Asia Pacific Conference on Circuits and Systems '96, Seoul, South Korea, 1996.
[25] Martin Vetterli and Henri J. Nussbaumer, "Simple FFT and DCT Algorithms with reduced number of operations," *Signal Processing Elsevier Science Publishers B.V. (North-Holland)*, vol. 6, pp. 267 - 278, 1984.
[26] Keun-Sup Lee, Young Cheol Park, and Hee Youn, "Software Optimization of the MPEG-Audio Decoder Using a 32-Bit MCU Risc Processor," *IEEE Transactions on Consumer Electronics*, vol. 48, pp. 671 - 676, 2002.
[27] Zhongde Wang, "Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 32, pp. 803 - 816, 1984.
[28] Naoki Suehiro and Mitsutoshi Hatori, "Fast algorithms for the DFT and other sinusoidal transforms," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 34, pp. 642 - 644, 1986.
[29] Mat Hans, "An MPEG Audio Decoder based on 16-bit Integer Arithmetic and SIMD Usage," pp. 463 - 468, 1997.
[30] Sungwook Yu and Earl E. Swartzlander Jr., "DCT Implementation with Distributed Arithmetic," *IEEE Transactions on Computers*, vol. 50, pp. 985 - 991, 2001.
[31] Patrick De Smet and Ignace Bruyland, "Optimised Recursive Subband Synthesis Windowing for Implementing Efficient MPEG Audio Decoders," *IEEE Signal Processing Letters*, vol. 10, pp. 303 - 306, 2003.
[32] http://www.mp3-tech.org/, http://www.mp3-tech.org/, http://www.mp3-tech.org/.

# Appendix A.

For the following Appendices please refer to corresponding files or directories on the accompanying CD.

**Appendix 1-1** clk_div.vhd

**Appendix 1-2** reader_rom.vhd

**Appendix 1-3** par_2_ser.vhd

**Appendix 1-4** header.vhd

**Appendix 1-5** bit_lut.vhd

**Appendix 1-6** valid_lut.vhd

**Appendix 1-7** N_lut.vhd

**Appendix 1-8** Layer_1_dec.vhd

**Appendix 1-9** Layer_1_dec_main.vhd

**Appendix 1-10** ext_lut.vhd

**Appendix 1-11** nb_lut.vhd

**Appendix 1-12** scale_lut.vhd

**Appendix 1-13** mult_unit.vhd

**Appendix 1-14** synth_filter.vhd

**Appendix 1-15** Nik_cont.vhd

**Appendix 1-16** Nik_lut.vhd

**Appendix 1-17** Nik_rom_def.vhd

**Appendix 1-18** Di_cont.vhd

**Appendix 1-19** Di_lut.vhd

**Appendix 1-20** Di_rom_def.vhd

**Appendix 1-21** mpeg_control_variables.vhd (file that sets the bit length throughout the VHDL model)

**Appendix 2** decode.c (the only file modified from the Standard C model)

**Appendix 3-1** scale_factors and nb mult.xls (spreadsheet to change bit lengths of `nb`, and `scalefactor`)

**Appendix 3-2** nik.xls (spreadsheet to change bit lengths of Nik filter coefficient)

**Appendix 3-3** di.xls (spreadsheet to change bit lengths of Di filter coefficient)

**Appendix 4** bit lengths.xls (spreadsheet to calculate bit lengths of VHDL model)

**Appendix 5** fl4.mp3 (test waveform from the standard [2])

**Appendix 6** Verification_data (this directory is the test data that was generated from the VHDL model to test compliance of the model, and includes the spreadsheet used to compare the data)